SCALABLE AND RESILIENT SPARSE LINEAR SOLVERS

A Dissertation
Presented to
The Academic Faculty

By

Piyush K. Sao

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

August 2018

SCALABLE AND RESILIENT SPARSE LINEAR SOLVERS

Approved by:

Dr. Richard Vuduc, Advisor
School of Computational Science and Technology
Georgia Institute of Technology

Dr. Xiaoye Sherry Li
Computational Research Division
Lawrence Berkeley National Laboratory

Dr. Haesun Park
School of Computational Science and Technology
Georgia Institute of Technology

Dr. Edmond Chow
School of Computational Science and Technology
Georgia Institute of Technology

Dr. Hao-Min Zhou
School of Mathematics
Georgia Institute of Technology

Dr. Umit Catalayurek
School of Computational Science and Technology
Georgia Institute of Technology

Date Approved: April 18, 2018

Dedicated to my parents

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Richard Vuduc for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

I would like to thank Dr. Xiaoye Sherry Li (LBNL), who has been very closely advising me throughout my thesis research. My internship at LBNL under her supervision was the stepping stone of bulk of the work covered in this thesis. During my internship, she introduced me to the area of the sparse direct solver. Throughout the thesis research, she has been very patient to answer my questions and, and provided me with any resource required for research.

I would like to thank my thesis committee: Dr. Edmond Chow, Dr. Haomin Zhou, and Dr. Haesun Park, and Dr. Umit Catalyurek for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives. I would also like to express my gratitude to Dr. Mikhail Smelyanskiy, for guiding me during my internship at Parallel Computing Lab, Intel Labs.

Among senior Ph.D. students who I had the opportunity to work with, I would like to thank Xing Liu, Oded Green, Ramakrishnan Kannan, "Ramki" and Aparna Chandramowlishran for being there to answer my questions and give me suggestions on every aspect of Ph.D. life. Working with them was a great learning opportunity for me. I would like to thank Ramki for helping me navigate through never ending mazes of Ph.D. life, pushing me when I was losing steam, calming me when I was exasperated. I would also like to thank Dr. Jeff Young for his valuable support and help with research, writing and preparing talks.

I would like to thank my colleagues Jiajia, Eswar, Oguz, Aaftab, Zhaomin, and Mohan for all the memorable time spent together in chasing deadline, at conferences, or during coffee break discussions. I would like to thank my friends outside of lab Prasun, Robert, Prateek, Suvadeep, Marat, Ravi, Sabyasachi, Nimit, Abhinav, Nishant, Arthy, Billbang, Pramod, Angelika, Ekin for making my Atlanta stay memorable. Memories of this part of the journey will always be very close to my heart. Finally, I would like to thank my friends from India Bhumika, Mamta, Rakhi, Parul, Amit, Abhimanyu, Mayank, and Harshit for still keeping in touch despite their busy lives.

I want to thank my parents and my elder brother for their love and support throughout my long graduate study. They have sacrificed greatly—not only during my time as a graduate student but since the day I was born-to make my life better. I would like to thank my younger brother Ankur for his enthusiastic support and affection, and my cousins Shreya, Aayush, Utkarsh, Reecha for their love and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis presents new algorithmic techniques that improve the scalability of the sparse linear solvers. Solving a system of linear equations which is sparse and large is one of the most frequently occurring subproblems in many areas of scientific computing.

We specifically address the two key issues for improving the scalability of sparse solvers. The first is reducing communication. Modern computing architectures have several levels of parallelism, including instruction level parallelism, thread level parallelism, hardware accelerator based parallelism and node level parallelism. Each level of parallelism has involves its own set of the memory hierarchy. Since the unit cost of data movement is significantly higher than the unit cost of a floating-point operation, to achieve good performance it becomes essential to reduce the data transfer between different levels of parallelism and also individual memory hierarchies within each level of parallelism. We consider these challenges in the context of the distributed memory sparse direct solver. Our baseline distributed memory sparse direct solver is SUPERLU_DIST, which only uses MPI for distributed memory parallelism, and otherwise relies on efficient underlying libraries, such as the Basic Linear Algebra Subroutines (BLAS), for additional performance and scaling.

In the first part of this work, we focus on improving the single node performance using hybrid programming and by exploiting on-node GPU and Xeon-Phi accelerators. The first main algorithmic contribution is a novel offload algorithm, which we call HALO, that targets heterogeneous architectures. The term HALO stands for highly asynchronous and lazy offload. The effect of the HALO algorithm is to reduce the communication between a host processor (CPU) and on-node accelerator.

In the second part of this work, we focus on improving the scalability of the sparse direct solver in the limit of a very large number of MPI processes. At such scales, it

is not possible to hide the cost of communication by overlapping or pipelining techniques. We specifically propose a new communication-avoiding sparse LU factorization algorithm. We refer to this algorithm as a "3D" sparse LU as it arranges the MPI processes in a logical three-dimensional grid, in contrast to the usual two-dimensional grid that is the state-of-the-art approach. The 3D sparse LU factorization algorithm reduces communication by aggressively exploiting elimination tree parallelism and using data replication, which maps naturally to a 3D arrangement of MPI process grids. We analyze the communication and latency costs for matrices arising from the numerical solution of two-dimensional and three-dimensional partial differential equations. Our analysis shows, for instance, that on planar sparse matrices of dimension $\mathcal{O}(n)$ the 3D algorithm reduces the asymptotic communication costs by a factor of $\mathcal{O}\left(\sqrt{\log n}\right)$ and latency by a factor of $\mathcal{O}(\log n)$. Overall, our on-node modification leads a performance improvement of up to 3× on different heterogeneous architectures. And inter-node modification results in 27× performance improvement for planar, and 2.5 × in case of non-planar sparse matrices.

Looking forward to future systems, we address a second key issue, which is that of resiliency. In next-generation computers with billions of computing elements, it is said that hardware faults will become the norm rather than an exception. A particularly insidious manifestation of a hardware fault is silent data corruption, where an undetected fault corrupts the state of a computation. We apply the principle of self-stabilization to construct fault-tolerant iterative linear solvers that can overcome such data corruptions by design. Informally, a system is said to be self-stabilizing if, starting from an arbitrary state, it comes to a valid state in a finite number of steps. We give two examples of iterative linear solvers, namely, steepest descent and conjugate gradient, to show how we can use the abstraction of valid and invalid states to construct fault-tolerant versions of these solvers. Our self-stabilized conjugate gradient algorithm can tolerate a very high fault rate with only a modest degradation in

its convergence rate.

CHAPTER 1

INTRODUCTION AND MOTIVATION

1.1 Problem statement

We are interested in solving a system of linear equations, $Ax = b$, where the matrix A is very large and sparse, as quickly as possible. Solving such a system directly affects our ability to conduct large-scale scientific simulations with greater accuracy and precision, which in turn accelerates the process of scientific discovery. *Sparse linear solvers* collectively describe the algorithms and software that solves such a system. Our approach for solving sparse systems is to exploit massively parallel computing systems to reduce the time to solution. In this thesis, we discuss current and future challenges that we face when we try to exploit such computing systems for general sparse solvers. At the time of this writing, the imminent scaling targets are so-called exascale supercomputers, which would be capable of performing 10^{18} floating-point operations per second, or 1 exaFLOP/s.

Scaling a sparse linear solver to an exascale supercomputer is challenging due to myriad issues. First, an exascale supercomputer offers a very high degree of parallelism. A large proportion of this parallelism comes from computational co-processors, or accelerators, today embodied by graphics processing units (GPUs) and Xeon-Phi co-processors. Algorithms with irregular memory access patterns, of which sparse linear solvers are an example, suffer from high communication overhead and might not effectively utilize accelerators. In addition, at this scale, data movement costs become the dominant problem. Further, in exascale machine with billions of components, erroneous execution of hardware are expected to become a norm rather than exceptions. Present day parallel algorithms do not address these problems.

Therefore, it is necessary to develop algorithms that can reduce communication overhead, utilize accelerators effectively, and are resilient to hardware faults. Further, given the current trend in high performance computing towards systems with increasing core counts, such algorithms will become a requirement for efficient solution of scientific problems in the near future.

Broadly, we can divide the contribution of the thesis into two parts. In the first part, we discuss how to exploit heterogeneous computing systems and reduce intra-node and inter-node communication to improve the scalability of *sparse direct solvers*. In the second part, we discuss the issue resilience in the future computing architecture and how does it affect the scalability of *sparse iterative linear solvers*. We describe *self-stabilizing* variants of such solvers, which can give a correct answer even in the presence of temporary faults, such as bit flips.

1.2 Contributions

1.2.1 Accelerated and Scalable Sparse Direct Solvers

We explore how to effectively exploit intra-node co-processors, or accelerators, in distributed memory sparse direct solvers. Such co- processors, by which we mean GPUs and Xeon-Phis, are widely deployed because of their attractive energy-efficiency characteristics. However, the challenge they pose is that distributed sparse direct solvers have complex data dependencies, irregular memory accesses, and highly variable arithmetic intensity. Thus, it is unclear a priori whether or by how much a distributed sparse direct solver can gain from the use of such co-processors [1].

Offloading Basic Linear Algebra Subprograms (BLAS) computations to a co-processor. We attempt to utilize co-processors for accelerating BLAS computations. Co-processors such as GPUs and Xeon-Phi have proved their efficacy for accelerating dense BLAS computation, and such computation can account for up to

70% of the time in the most expensive step, *numerical factorization* in distributed sparse direct solvers. Thus, offloading BLAS computations is a natural first step towards this goal.

The BLAS computation within a distributed sparse direct solver is dominated by many calls to GEMM (dense matrix multiply) on small and irregular sized operands. Since each offload to a co-processor incurs a non-trivial latency cost, directly offloading such calls may not give any performance improvement. We overcome this issue by aggregating several small BLAS calls into a few bigger ones, and then apply latency-hiding strategies such as software pipelining. Our approach gives us a performance improvement of up to 3×.

Reducing intra-node communication. We identify two key limitations of only offloading BLAS computations to the GPU. First, once the BLAS calls are accelerated, the so-called SCATTER computation becomes the performance bottleneck. Secondly, the slowest link in the BLAS offload approach is the data transfer between the host (CPU) processor and any co-processors via Peripheral Component Interface Express (PCIe), and in newer computing systems this cost becomes dominant. That, in turn, potentially eclipses any performance gains from BLAS offload. To overcome this problem, we present a new algorithm for accelerating distributed sparse direct solver that we call the HALO algorithm. The name is a shorthand for *highly asynchronous lazy offload*; it refers to the way the algorithm combines the highly aggressive use of asynchrony with the accelerated offload, lazy updates, and data shadowing (*a la* halo or ghost zones), all of which serve to hide and reduce communication, whether to local memory, and over PCIe.

Reducing inter-node communication. Lastly, we explore how to effectively utilize a large number of distributed nodes to reduce the time to solution for the sparse direct solver. The challenge is again the relative cost of data transfer between dif-

ferent computing nodes increases as we try to use a large number of nodes. At such scales, previous approaches to hide communication by overlapping or pipelining is not effective anymore. We propose a new algorithm that we call three-dimensional sparse LU factorization algorithm, called so to reflect the three dimensions logical arrangement of processes. We combine the principles of data replication with so-called *tree-parallelism* to reduce the communication. Our performance models suggest that we can reduce the communication by a factor of $\mathcal{O}(\log n)$ for planar matrices derived from a discrete $n \times n$ domain; and by a constant factor for non-planar matrices. Overall, our approach leads to speed-up of up to 28× while effectively utilizing 24,000 cores of a Cray XC30 machines.

1.2.2 Self-stabilizing iterative solvers.

A fault is an instance of computing hardware deviating from its expected behavior. Traditionally, we design algorithms assuming that the underlying computing machine is reliable, meaning we ignore the possibility of any faults. However, the probability of a fault increases proportionally to the number of computing elements. Therefore, when we are trying to scale computation to an extremely large number of computing elements, the reliability of the system decreases.

A particularly insidious type of hardware fault is *silent data corruption*, where a fault causes a corruption in the data that is not reported to the user. Such a corruption would propagate to most of the intermediate variables, and eventually result in an incorrect result. In the case of sparse iterative solvers, such corruptions may result in a solution that does not satisfy the desired precision yet may often be reported as having converged to the desired accuracy. Thus, a hardware fault can cause serious reliability issues when scaling to a large number of computing elements. We are interested in sparse linear solvers, which can converge to a correct solution even when underlying hardware is unreliable.

We propose to apply the principle of *self-stabilization* to construct fault-tolerant iterative linear solvers. Informally, a system is said to be self-stabilizing if starting from any state, valid or invalid, comes to a valid state in a finite number of states. The self-stabilization property provides a natural form of resilience against the class of transient (temporary) soft-faults. A number of algorithms, such as numerical fixed-point iteration, have the property of self-stabilization by design. However, more attractive Krylov subspace-based iterative solvers do not have such a property. Nevertheless, we show that an algorithm that is not naturally self-stabilizing can be modified so as to become self-stabilizing. In particular, we construct as a proof-of-concept self-stabilized versions of both the Conjugate Gradients (CG) and the Steepest Descent methods for solving symmetric positive definite systems.

Numerical experiments on self-stabilizing CG shows that it can tolerate extremely high fault injection rates. We further study the overhead of such methods analytically and by experiment.

1.3 Organization

This dissertation is organized as follows.

In Chapter 2, we briefly review sparse linear systems and solvers. We classify sparse linear systems based their sources and numerical properties. Additionally, we discuss solvers for different classes of sparse systems.

In Chapters 3 to 6, we discuss inter- and intra-node enhancements on distributed memory sparse direct solver for improving single-node performance and multi-node scalability. We start by discussing the relevant theory of sparse direct solvers in Chapter 3. We use SUPERLU_DIST as the baseline over which we implement our proposed algorithms. We present the relevant details of SUPERLU_DIST in Chapter 3.

Chapter 4 presents the first CPU-GPU distributed memory sparse direct solver.

It explores how GPUs can be used as BLAS *accelerator*. In addition to GPU acceleration, we discuss how a different hybrid parallel scheme that combines MPI, OpenMP, and GPU acceleration, both implicitly and explicitly, affect the performance of a distributed memory sparse direct solver overall.

Chapter 5 presents the new HALO algorithm for exploiting on-node accelerators. Our focus is on reducing intra-node communication, which becomes a bottleneck for newer architectures. We also move away from considering only GPUs as co-processors, generalizing the technique to include Intel Xeon-Phi as another co-processor type. We implement the HALO for Intel Xeon-Phi accelerated heterogeneous clusters and discuss several optimization techniques. We later implement the HALO for GPU-based systems as well, and share our experience with using both GPUs and Xeon-Phi as co-processors for accelerating SUPERLU_DIST.

In Chapter 6, we move away from single-node optimizations, and focus on improving the scalability of SUPERLU_DIST on a large number of nodes. We present a new algorithm, namely, a three-dimensional (3D) LU factorization algorithm for sparse matrices, which improves the strong and weak scalability of SUPERLU_DIST on a large number of nodes. Further, we construct performance models for understanding performance scalability given different types of matrices. We also present the results of strong and weak scaling experiments on different numbers of processors.

In Chapters 7 and 8, we consider resilient computation and present the principle of self-stabilization for constructing fault-tolerant iterative solvers. We start with a general discussion of faults in computing systems and motivate the need for new algorithmic resilience techniques. Then, we discuss the principle of self-stabilization and how we can look at iterative solvers as systems with states and state-transition rules. We explain the idea of augmenting an iterative algorithm with a periodic correction scheme that makes it self-stabilizing. We then give examples of two self-stabilizing algorithms, one for the steepest descent method and

the other for the conjugate gradient method, using the principle of self-stabilization. Lastly, we present empirical experiements and analytical models to estabalish the efficacy of self-stabilizing conjugate gradient algorithm in presence of simulated injected faults.

Chapter 9 concludes, looking specifically into newer problems that one might expect to arise in the next generation of high-performance computing systems.

CHAPTER 2

SPARSE LINEAR SYSTEM AND SOLVERS

2.1 Sparse System of Linear Equations

In this chapter, we provide a brief overview of system of linear equations, their classification and methods to solve them. A system of linear equations in n variables, $x_1, x_2, \ldots\ x_n$, is represented in terms of scalars as follows:

$$\begin{array}{lllll} a_{11}x_1+ & a_{12}x_2+ & \ldots & a_{1n}x_n & = b_1 \\ a_{21}x_1+ & a_{22}x_2+ & \ldots & a_{11}x_n & = b_2 \\ \vdots & \vdots & \vdots & \vdots & = \vdots \\ a_{n1}x_1+ & a_{n2}x_2+ & \ldots & a_{nn}x_n & = b_n \end{array} \tag{2.1}$$

This can be also represented in matrix form as:

$$Ax = b, \tag{2.2}$$

Where $A = [a_{ij}]_{i,j}$ is called the coefficient matrix and the vector $b = [b_i]_i$ is called the right hand vector. We call any algorithm for solving this system a *linear solver*.

Definition 1 (Error)**.** Let x^* be the exact solution, $Ax^* = b$. Then for any candidate solution, x, the difference $x - x^*$ is defined to be the *error* in the solution.

Definition 2 (Residual)**.** The quantity $b - Ax$ is defined to be the *residual* vector.

For the exact solution, both the error and the residual are zero. However, since we do not have access to the exact solution, we cannot calculate the error for a given candidate solution x. Instead, the accuracy of the solution is typically measured with the norm of the residual.

A matrix is a *sparse* if most of the entries of A are zero. Linear systems from many scientific application are extremely sparse, with a typical number of nonzeros growing like $\mathcal{O}(n)$ rather than n^2. To be able to solve very large problems, it becomes essential to exploit sparsity both with respect to storage and the number of operations. Depending on the source of the problem, the matrix may exhibit many different types of sparsity *patterns*. Specialized linear solvers can exploit specific patterns. Other linear solvers may exploit linear-algebraic properties of the coefficient matrix. In some cases, the right hand sides can also be sparse and linear solvers may try to exploit sparsity pattern in b as well.

2.2 Classification of Sparse Systems

In this dissertation, our discussion is limited to sparse matrices where non-zeros belong to the field of real numbers. However, most of this discussion also applies to complex matrices, though for simplicity, we will use exclusively terms relevant to real-valued problems (e.g., we will use the term *symmetric* instead of *Hermitian*).

2.2.1 Based on Source Application

Numerical solution of partial differential equations (PDEs). Finding the numerical solution to boundary value partial differential equations (PDE) is a major source of sparse linear systems. Such PDEs involve only spatial differentials (or integral) operators, but no time derivatives. In the physical sciences, their dimension is usually limited to one, two, or three dimensions; however, there are some specific applications where the dimension can be higher than 3, including relativistic mechanics and quantum mechanics.

A canonical problem is the *1D Laplace equation*,

$$\frac{\partial^2 u}{\partial x^2} = f(x). \tag{2.3}$$

A uniform finite-difference discretization would lead to the equations,

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = f_i, \tag{2.4}$$

where h is the grid spacing. In matrix form, Equation (2.4) becomes

$$\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix} . \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix} . \tag{2.5}$$

Let's denote the coefficient matrix in the 1D case by $K_{1\mathrm{D}}$. Then the coefficient matrix for the 2D Laplace equation, denoted $K_{2\mathrm{D}}$, can be written in terms of $K_{1\mathrm{D}}$ as follows:

$$K_{2\mathrm{D}} = \begin{bmatrix} K_{1\mathrm{D}} + 2I & -I & & & & \\ -I & K_{1\mathrm{D}} + 2I & -I & & & \\ & -I & \ddots & \ddots & & \\ & & \ddots & \ddots & -I & \\ & & & -I & K_{1\mathrm{D}} + 2I \end{bmatrix} ., \tag{2.6}$$

where I is the identity matrix. Similarly, $K_{3\mathrm{D}}$ can be written in terms of $K_{2\mathrm{D}}$ as follows:

$$K_{3\mathrm{D}} = \begin{bmatrix} K_{2\mathrm{D}} + 2I & -I & & & & \\ -I & K_{2\mathrm{D}} + 2I & -I & & & \\ & -I & \ddots & \ddots & & \\ & & \ddots & \ddots & -I & \\ & & & -I & K_{2\mathrm{D}} + 2I \end{bmatrix} .. \tag{2.7}$$

Implicit schemes for time-dependent partial differential equations. Another important source for sparse linear systems is implicit schemes for time-dependent

PDEs. For instance, consider the following time-dependent heat equation ,

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2}. \tag{2.8}$$

Applying the Crank-Nicholson scheme to discretize this system in space and time would lead to following equations:

$$\frac{u_i^{n+1} - u_i^n}{h} = \frac{c}{2k^2}\left((u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}) + (u_{i+1}^n - 2u_i^n + u_{i-1}^n)\right); \tag{2.9}$$

where $h = \Delta x$ is the grid spacing and $k = \Delta t$ is the time step size. Let $r = \frac{2ck}{h^2}$. Then, this system may then be written as follows:

$$-ru_{i+1}^{n+1} + (1+2r)u_i^{n+1} - ru_{i-1}^{n+1} = ru_{i+1}^n + (1-2r)u_i^n + ru_{i-1}^n \tag{2.10}$$

which is similar to Equation (2.4) in that it has the form of a tridiagonal matrix.

Linear and non-linear programming. Non-linear programming often involves solving large systems of linear equations, which are not necessarily extremely sparse. For instance, consider following constrained quadratic programming problem:

$$\min f(x) = \frac{1}{2}x^T Bx - x^T b; \tag{2.11}$$

$$\text{Subject to } Ax = c. \tag{2.12}$$

Equation (2.11) reduces to following linear system of equations:

$$\begin{bmatrix} B & A^T \\ A & 0 \end{bmatrix} = \begin{bmatrix} b \\ c \end{bmatrix}. \tag{2.13}$$

Equation (2.13) is called the *Karush-Kuhn-Tucker* (KKT) system. We include

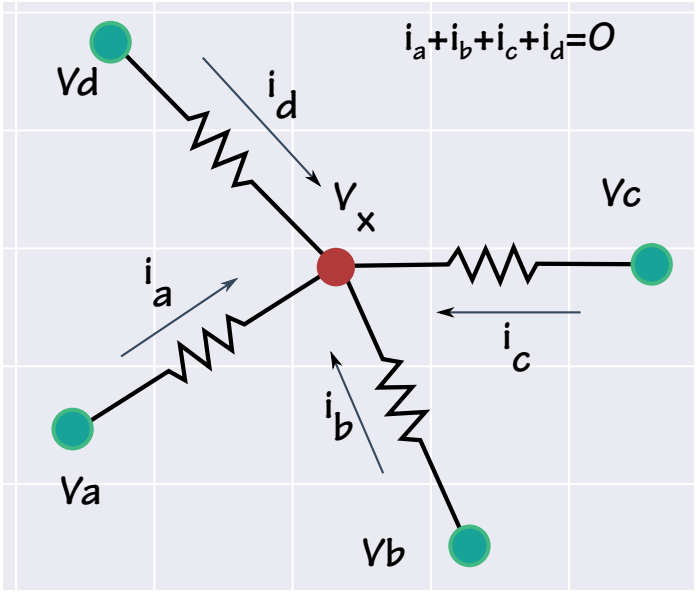


Figure 2.1: Kirchoff's Current Law on resistive networks

KKT systems from PDE optimization in our collection of test problems [2].

Non-linear systems of equations. Similarily, solving system of non-linear equation using Newton's iteration often involves applying inverse of the Jacobian matrix.

Circuit and network analysis. The analysis of electrical circuits, such as a DC circuit network that contains only passive elements or a quiescent point analysis of non-linear circuits, often leads to Kirchoff's current law or voltage law formulations like those shown in Figure 2.1. Kirchoff's current law states the sum of all the incoming currents to a node is zero. The current between any two nodes a and b connected by an Ohmic conductor with conductance G_{ab} is given by

$$\vec{i}_{ab} = G_{ab}\{V_a - V_b\}.$$

Applying KCL on Figure 2.1 results in the following equation:

$$G_{ax}\{V_a - V_x\} + G_{bx}\{V_b - V_x\} + G_{cx}\{V_c - V_x\} + G_{dx}\{V_d - V_x\} = \quad 0; \quad (2.14)$$

$$G_{ax}V_a + G_{bx}V_b + G_{cx}V_c + G_{dx}V_d - (G_{ax} + G_{bx} + G_{cx} + G_{dx})V_x = \quad 0. \quad (2.15)$$

Kirchhoff's current law leads to a symmetric sparse linear system (Equation (2.15)) where the sum of coefficients in the equation is zero. Note the finite difference discretization of the heat equation also lead to a structurally similar set of equations. More generally, any physical system where the so-called *flux* is conserved would yield such forms. Specialized linear solvers such as [3], exists for solving system of linear equation arising from circuit simulation.

2.2.2 Based on Numerical and Linear Algebraic Properties of the Matrix

Symmetry. A matrix can be symmetric $A = A^T$, antisymmetric (or skew-symmetric) $A = -A^T$, or asymmetric when it is neither of the two. Symmetric matrices can be further classified based on their eigenvalue spectrum. A symmetric matrix A is *positive definite* if all its eigenvalues are positive; *semidefinite* if they are non-negative; and *indefinite* if the set of eigenvalue contains both positive and negative elements.

Geometry. A sparse matrix may also be categorized based on the dimension of its associated graph. The dimension of a graph is the least possible dimension of a Euclidean space where the graph can be embedded such that each edge has a unit length and there are no edge crossings. Sparse matrices coming from finite difference discretizations of 2D and 3D PDEs have graph dimensions of two and three, respectively. Typically, the higher its graph dimension, the denser a sparse matrix becomes.

Numerical conditioning. The condition number of floating-point computation is a bound on maximum change in the output for any change in the input with a unit norm. For solving $Ax = b$, the condition number in the two-norm of the problem is the ratio of largest and smallest singular value and is denoted by $\kappa(A)$. A higher condition number reflects the inherent difficulty in solving the system.

2.3 Overview of Sparse Linear Solvers

2.3.1 Stationary Iterations

Stationary iterations are one of the most basic numerical methods for solving a system of linear equations. Such a method constructs a sequence $< x_k >$ that converges to the solution of $Ax = b$. It constructs an element x_{k+1} of the sequence by applying a linear transformation on previous element, x_k. The transformation does not change as iteration progresses. Typically, a stationary iteration may be expressed as a sum-decomposition of the matrix $A = E + F$ such that computing $x_{k+1} = E^{-1}(b - Fx_k)$ is computationally easy. For instance, in *Jacobi iterations*, the E matrix is the diagonal of A; and in *Gauss-Seidel iterations*, the E matrix is an upper or lower triangular submatrix of A. Stationary iteration by itself converges very slowly. Therefore, it is rarely used as a standalone solver. Often, it is used in conjunction with other solvers or as a preconditioner.

2.3.2 Krylov-subspace Iterations

For a given matrix A and a right-hand side b, the *Krylov subspace*, $\mathcal{K}(A, b)$, is defined as follows:

$$\mathcal{K}(A, b, m) = \{b, Ab, A^2b, \ldots A^{m-1}b\} \tag{2.16}$$

A Krylov subspace method implicitly or explicitly constructs the Krylov subspace.

For a given matrix A and a right-hand side b the Krylov subspace $\mathcal{K}(A,b)$ is defined as follows:

$$\mathcal{K}(A,b) = \{b, Ab, A^2b, \ldots\} \tag{2.17}$$

A Krylov subspace method implicitly or explicitly constructs the Krylov subspace. In every iteration, it constructs an approximate solution $x_k \in \mathcal{K}(A,b)$, so that x_k is optimal in some specified sense. For instance, the *generalized minimal residual* (GMRES) method constructs x_k to minimize the 2-norm of the residual, $r_k = b - Ax_k$. When the matrix is symmetric and positive definite, then one can use a specialized Krylov subspace based called Conjugate Gradients (CG), which minimizes $e_k^T A e_k$, where e_k is the error vector $e_k = x^* - x_k$.

In exact arithmetic, Krylov subspace methods converge in n steps, where n is the order of the matrix. Typically, computing up to n iterations is computationally very expensive and numerically unstable. Therefore, Krylov subspace methods are used in practice as iterative methods and, typically, require far fewer than n iterations to reach the required precision. The convergence of the Krylov subspace methods depends strongly on the condition number of A. Typically, some type of preconditioning is essential for these methods to converge within reasonable number of iterations.

2.3.3 Direct Methods

A *direct method* solves the system of linear equations by directly applying the A^{-1} operator on the vector b. However, it is rare to calculate A^{-1} explicitly. Instead, the matrix A is usually factored into the product of 2 or more matrices such that applying inverse of each factor is cheap. For instance, in the LU-decomposition, the matrix A is factored into the product LU, where L is a unit lower triangular matrix and U is a upper triangular matrix. It then applies the operator A^{-1} as $U^{-1}L^{-1}$. This method works because the inverse of a triangular matrix can be easily applied by so-called

forward or *backward* substitution methods.

When the matrix is symmetric and positive definite, then we can use *Cholesky* algorithm to construct a factorization LL^T, which requires fewer floating point operations than computing an LU decomposition. The QR decomposition is another popular technique, particularly for overdetermined systems, where the matrix A is decomposed into an orthonormal matrix Q and a upper triangular matrix R.

2.3.4 Multigrid Methods

Multigrid is a popular method for solving sparse systems arising from certain types of PDEs. This method is based on the observation that stationary iterations are typically good at reducing errors of high frequency, but not good at reducing low-frequency components of the error. However, these low-frequency errors on a grid at a particular resolutoin resemble high-frequency errors at a coarser discretization. Multigrid methods exploit this phenomenon by running stationary iteration running at grids with different coarsening levels to effectively reducing all frequency components of the error. Multigrid methods have linear space and storage requirement, and are, therefore appealing wherever applicable. When the linear system of equations is not from a geometric PDE source, variants of multigrid such as algebraic multigrid can be still be used.

2.4 Conclusion

Sparse linear solvers are one of the most fundamental kernels in scientific computing. While there are many classes and subclasses of sparse linear solvers, Gaussian elimination based direct solvers and Krylov subspace based iterative solvers are among the most widely used. However, scaling these direct and iterative solvers on exascale machines is far from a trivial task due to challenges such as power, reliability, and the need to exploit massive parallelism. In this dissertation, we will

address three key issues that the prior art had not addressed sufficiently, namely, a) exploiting co-processors acceleration for sparse direct solvers; b) reducing inter-node communication of sparse direct solvers; and c) improving the algorithmic resilience of iterative solvers.

CHAPTER 3

DIRECT SOLVERS FOR SPARSE MATRICES

Direct methods solve a system of linear equations by Gaussian Elimination, also known as row reduction. In matrix form, Gaussian elimination is equivalent to factoring a matrix A into the product LU, where L is a unit lower triangular matrix and U is an upper triangular matrix. When A is symmetric and positive definite (SPD), one can factor it into LL^T using a Cholesky factorization algorithm. In this dissertation, our discussion is limited to general LU factorization. However, much of the discussion applies to Cholesky factorization as well.

Sparse direct solvers tend to be considerably more complex than their counterparts for dense matrices. The main difficulty arises due to "fill-in" of the sparse matrix A during the factorization process. The amount of fill depends on the non-zero pattern of A. Managing fill-in adds engineering difficulties such as memory management and complex data dependencies, which are not present in a dense solver.

In light of these challenges, a typical sparse direct solver consists of the following four distinct components:

- *Fill-reducing permutation*: The matrix A is permuted to reduce fill-in of the L and U factors.
- *Symbolic factorization*: The (permuted) matrix is analyzed to estimate the non-zero structure of the factored matrix, which allows for preallocation of memory to hold the output.
- *Numerical factorization*: This component calculates the numerical entries of the L and U factors.

- *Triangular solve*: Lastly, the solution x is computed, given L, U, and the right-hand side b.

Outline of this chapter. We review the theoretical and practical aspects of all the steps involved in a sparse direct solver. We start with a discussion of the graph representation of sparse matrices, including those tools from graph theory that are most relevant to the design of a sparse direct solvers. Followed that, we summarize the key algorithms and variants for all the steps involved in the direct solver. This discussion includes the design of current parallel direct solvers.

3.1 Preprocessing

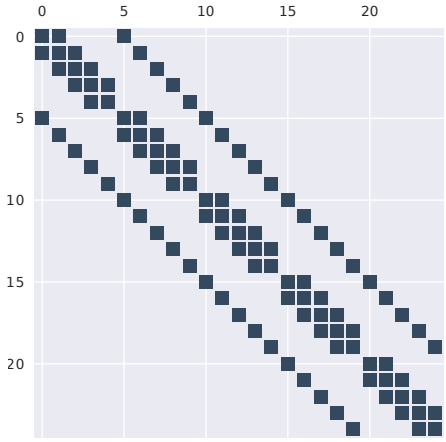
The preprocessing step serves three purpose. First, it determines the order of elimination of variables. Secondly, it performs a symbolic factorization to ascertain the non-zero structure of L and U factors and allocates memory for them in advance. Thirdly, it scales the matrix by multiplying by a diagonal matrix that increases the relative magnitude of diagonal entries, to improve numerical stability.

3.2 Graph Representation of a Sparse Matrix

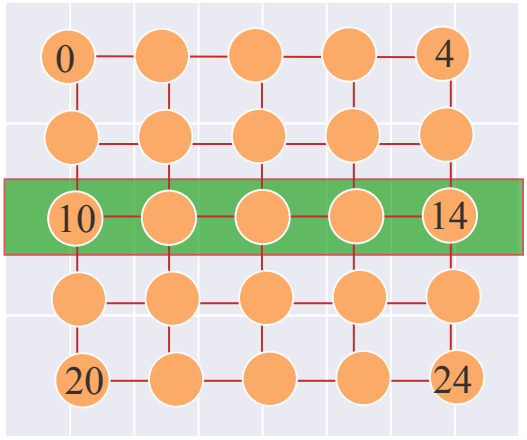
Much of the engineering that underlies the design of a sparse direct solver relies on the following key abstraction, which is the representation of a sparse matrix by a graph.

Definition 3 (Associated Graph of a Sparse Matrix)**.** For a given unsymmetric sparse matrix A of dimension n, a weighted directed graph $G = (V, E)$ is called the *associated graph of the sparse matrix A* if it satisfies the following properties:

1. G has n vertices numbered from 0 to $n-1$, $V = \{i | 0 \leq i < n,\ i \in \mathbb{Z}\}$;

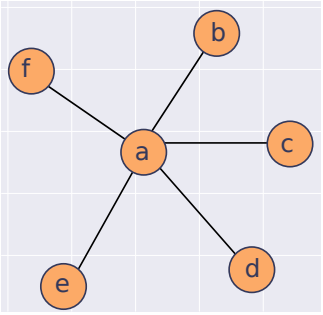


(a) A 5×5 sparse matrix



(b) The associated graph and separator

Figure 3.1: A sparse matrix (Figure 3.1a), its associated graph (Figure 3.1b), and a separator (highlighted in green).



(a) A "star" graph

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{bmatrix} x & x & x & x & x & x \\ x & x & & & & \\ x & & x & & & \\ x & & & x & & \\ x & & & & x & \\ x & & & & & x \end{bmatrix}$$

(b) Elimination order-1

$$\begin{array}{c} \\ b \\ c \\ d \\ e \\ f \\ a \end{array} \begin{bmatrix} x & & & & & x \\ & x & & & & x \\ & & x & & & x \\ & & & x & & x \\ & & & & x & x \\ x & x & x & x & x & x \end{bmatrix}$$

(c) Elimination order-2

Figure 3.2: Elimination orders for star graph

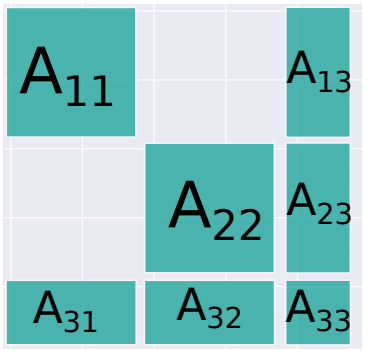
2. for every non-zero $(A)_{ij}$, G has an edge e_{ij} of weight $(A)_{ij}$ from the i-th vertex to the j-th vertex; and

3. if $(A)_{ij} = 0$ then there is no edge between vertex i and j in G.

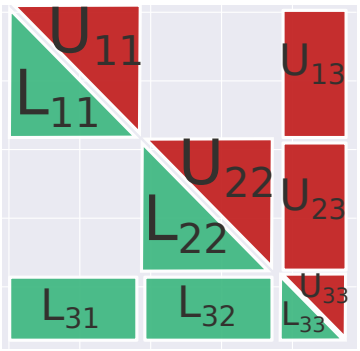
In Figure 3.1a, we show a pentadiagonal matrix , which might arise from a finite difference discretization of a PDE on a 2D square grid, as shown in Figure 3.1b.

3.3 Orderings that Reduce Fill-in

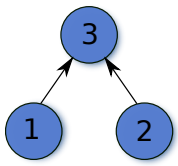
The order in which one eliminates variables will affect whether and where new non-zeros appear.



(a) A 3×3 block sparse matrix



(b) The L and U factors



(c) The elimination tree

Figure 3.3: In Figure 3.3a, we show the block sparse matrix A obtained from nested dissection of the adjacency graph of A. The L and U factors overwrite A after the factorization, as shown in Figure 3.3b. The elimination tree (Figure 3.3c) captures the dependencies between the factorization of A_{11}, A_{22}, and A_{33}.

For instance, consider the star matrix of Figure 3.2a and two possible orders for elimination. In the first elimination order, Figure 3.2b, all of the non-zero structure is destroyed after elimination of node-1. By contrast, in the second elimination order, no non-zero blocks will be introduced.

There are many algorithms to heuristically generate such a fill-in reducing ordering. These include variants of *minimum degree ordering*, orderings based on graph partitioning such as nested dissection ordering, and ordering that reduce the bandwidth of the matrix, to name a few. For conceptual understanding of how such orderings works, we describe the nested dissection method. Interested readers can find details of other ordering schemes elsewhere [4, 5].

3.3.1 Nested Dissection Ordering

Definition 4 (Vertex Separator)**.** A *vertex separator* (or just *separator*) S of a graph G is a subgraph that partitions G into three disjoint subgraphs, (C_1, S, C_2), so that C_1 and C_2 are disconnected.

A *good* separator is small and balances the partitions C_1 and C_2. In Figure 3.1b, we highlight a good separator in green.

In the nested dissection ordering of sparse matrices, we use a graph partitioning tool to find a separator S and, thereby, obtain the partitions C_1 and C_2. Using this partition, we order the sparse matrix A so that vertices in C_1 and C_2 come first, followed by the vertices in S. For instance, the block sparse matrix in Figure 3.3a shows one such ordering, where A_{11}, A_{22}, and A_{33} correspond to C_1, C_2, and S respectively, with remaining sub-matrices representing the edges that connect these partitions. Then, C_1 and C_2 can be recursively partitioned to get more disjoint subgraphs of A, a process known as *nested dissection* (ND). Graph partitioning tools like METIS can compute ND partitions [6].

Definition 5 (Fill-graph)**.** The fill-graph G^+ of a sparse matrix with associated graph G, is the associated graph of the factored sparse matrix $L+U$.

3.4 Elimination Tree

Once the order of elimination is determined, we can construct a tree of dependencies due to that order. This tree is the *elimination tree*, or *etree* for short. The etree is induced as follows: for each vertex v, its *parent*, $P(v)$, is the earliest node in the elimination order upon which v depends. Typically, a vertex will have only one parent but can have multiple children. Any vertex can be factorized only when all its children have been factored.

In the case of a nested dissection ordering, the parent of both partitions C_1 and C_2 is the separator S. Thus one level of the etree appears as shown in Figure 3.3c. When we further divide C_1 and C_2, we get an etree with multiple levels as shown in Figure 3.7b. Typically, the nested dissection ordering leads to binary trees. However, some graphs can have multiple disjoint components, in which case the etree is actually a forest.

3.5 Supernodal Associated Graph

Many sparse matrices have dense substructures, which can be exploited for cache-efficient computation. The associated graph of such matrices shows community-like structures, namely, cliques. Also, the associated graph can typically be partitioned into groups of vertices that are densely connected.

A *supernode* is a set of vertices in the associated graph[1]. As a part of pre-processing supernodal structure uis extracted. To obtain the supernodal associated graph, we replace the all the nodes from a supernode in the associated graph with a single node, while keeping the connectivity.

3.6 Numerical Factorization

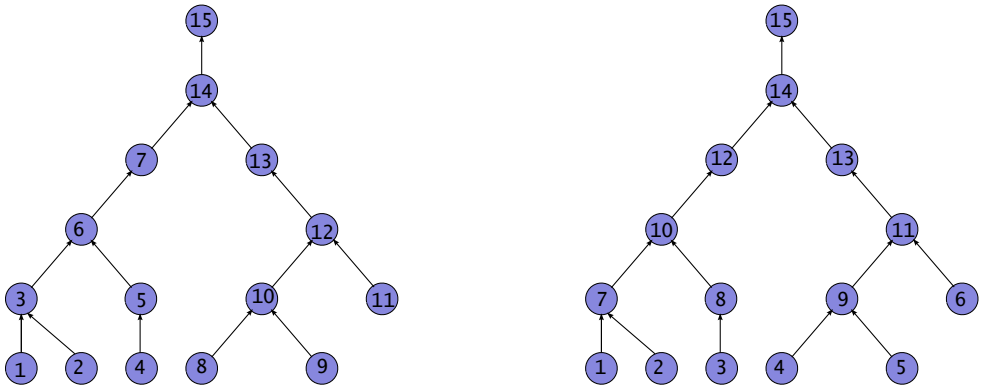
Consider the LU factorization of the 3×3 block sparse matrix shown in Figure 3.3a. The L and U factors are computed iteratively. There are 3 main steps involved in the factorization.

1. *Diagonal factorization*: $A_{ii} \rightarrow L_{ii}U_{ii}$
2. *Panel update*: $U_{ij} = L_{ii}^{-1}A_{ij}$ and $L_{ji} = A_{ji}U_{ii}^{-1}$
3. *Schur-complement update*: $A_{jk} = A_{jk} - L_{ji}U_{jk}$

3.6.1 Elimination Order

We show two possible way to traverse the etree in Figure 3.4. In a sequential factorization, there is no practical difference in the performance of either ordering. However, in a parallel factorization, the bottom-up ordering is advantageous as it exposes more parallelism since all leaves are independent.

[1] that are, typically, densely connected.



(a) Post order (b) Bottom-up order

Figure 3.4: Elimination order for sparse LU factorization

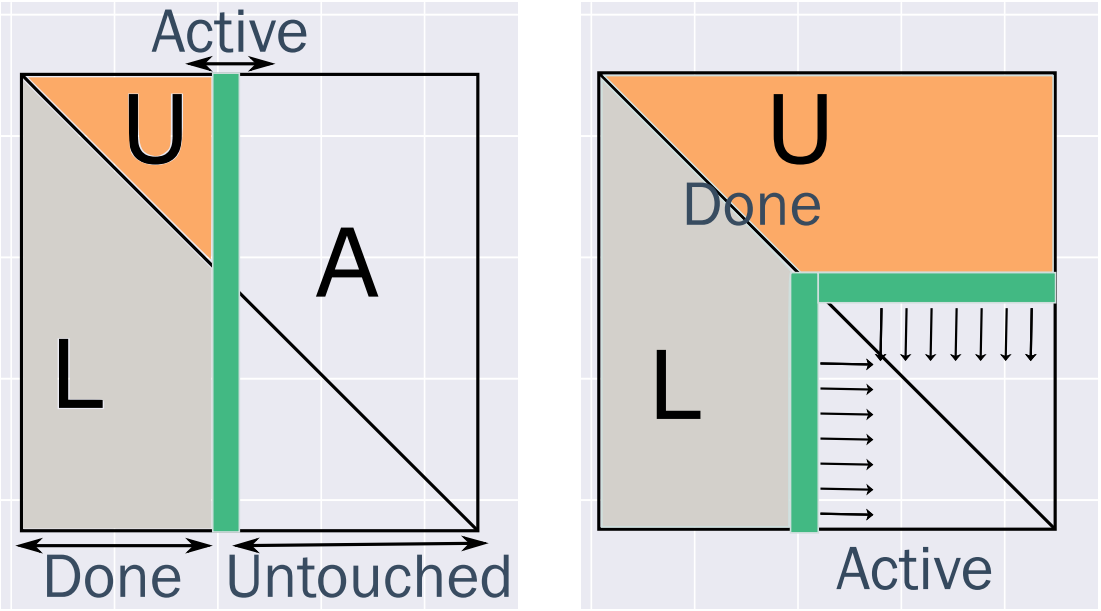
3.6.2 Scheduling of Operations

For a dense LU factorization, the operations can be arranged in the following two basic variations, based on when the k-th Schur-complement update is performed on a given block.

Right-looking (or eager scheduling). In a *right-looking* factorization, the Schur-complement update of the entire trailing matrix is performed right after the panel factorization, as illustrated in Figure 3.5b. Once the panels are computed and the Schur-update is done, these panels are no longer referred to until the triangular solve stage of the computation.

Left-looking (or lazy scheduling). In the *lazy* or *left-looking* scheduling, in the k-th iteration we only calculate the k-th column of the LU matrix and do *not* perform the Schur-update on rest of the trailing matrix. This scheme is illustrated in Figure 3.5a.

In the sparse case, we have another variant, which is called the *multifrontal method*.



(a) Left-looking LU factorization

(b) Right-looking LU factorization

Figure 3.5: Region of operations on left and right looking LU factorization

Multifrontal method. In the multifrontal method, for any node, the Schur-complement update is calculated right after panel factorization. However, these updates are not applied immediately to the destination blocks. Updates are typically stored in the form of dense matrices and, before any node is factorized, we assemble the updates from the all its children and then factorize it and keep its frontal matrix.

Discussion. The left-looking factorization incurs fewer writes and at the expense of a large number of reads. Therefore, it is beneficial on systems where writing is considerably more costly than reading. On the hand, the right-looking algorithm has a lot of parallelism in the Schur-complement.

The multifrontal method can exploit tree parallelism well, at the expense of extra memory. A more comprehensive discussion on the impact on performance of these different scheduling strategies can be found elsewhere [7, 8].

3.7 Triangular Solve

3.7.1 Forward and Backward Substitution

Forward substitution is an efficient method for solving a lower triangular system. A unit lower triangular matrix L_n, of order n, can be recursively described as follows:

$$L_n = \begin{bmatrix} 1 & 0 \\ l_0 & L_{n-1} \end{bmatrix}. \tag{3.1}$$

To solve $L_n x = y$, note that $x_0 = y_0$, so that

$$L_{n-1} x_{n-1} = y_{n-1} - x_0 l_0. \tag{3.2}$$

And the values of $x_1,\ x_2, \ldots x_{n-1}$ may be found recursively.

Backward Substitution is similar to forward substitution but is used for applying inverse of an upper triangular system. Since an upper triangular matrix can be transformed to a lower triangular matrix by reordering the variables in the reverse order, the backward substitution is equivalent to forward substitution on this transformed matrix.

3.7.2 Iterative Refinement

When the numerical factorization is not accurate enough, due, for instance, to the accumulation of floating-point errors, then we can refine the solution iteratively using the inaccurate factorization [9]. Algorithm 1 shows the basic procedure.

3.8 Overview of SUPERLU_DIST

SUPERLU_DIST performs a sparse LU factorization using the so-called *supernodal approach*. A *supernode* is a set of strongly connected vertices in the graph representation of the sparse matrix. During its preprocessing step, SUPERLU_DIST extracts

Algorithm 1 Iterative Refinement For Solving $Ax = b$

Require: $LU \approx A$, x_0, ϵ_0

1: $r \leftarrow b - Ax_0$
2: **while** $\|r\| \geq \|A\|_\infty$ **do**
3: $\quad d \leftarrow U^{-1}L^{-1}r$
4: $\quad x \leftarrow x + d$
5: $\quad r \leftarrow b - Ax$
return x

the supernodal structure from the input matrix, allowing it to store the sparse matrix as a collection of dense submatrices. This dense submatrix representation becomes the basis for exploiting fast level-3 BLAS operations, such as GEMM. However, unlike the case of factoring purely dense matrices, these dense subproblems have widely varying sizes.

3.8.1 Factorization algorithm

SUPERLU_DIST uses the Message Passing Interface (MPI) to express its distributed memory parallelism. The MPI processes are logically arranged in a two-dimensional (2D) process grid. On this 2D process grid, SUPERLU_DIST distributes the input matrix A in a block cyclic fashion. For example, Figure 3.7a shows a sparse matrix distributed among six (6) MPI processes, all arranged in a 2×2 grid.

3.8.2 SUPERLU_DIST Factorization kernels

SUPERLU_DIST factors each supernode sequentially. Factorization of the k-th supernode involves two main phases: panel-factorization and Schur-complement update. Both phases consist of many communication and computation kernels. We briefly describe the key kernels and their functions below. The interested reader will find a more comprehensive description of the factorization algorithm elsewhere [10].

- ***Panel-Factorization***

Algorithm 2 SUPERLU_DIST Sparse LU Factorization

Input: Distributed sparse matrix A;
On each MPI process p_{id} do in parallel:
for $k = 1,\ 2,\ 3\ldots\ n_s$ **do**
    Synchronize all processes
Panel Factorization
    if p_{id} owns $A(k,k)$ **then**
        Factor $A(k,k)$ and send $L(k,k)$ to $P_r(k)$ who need it
        Send $U(k,k)$ to $P_c(k)$
    if $p_{id} \in P_c(k)$ **then**
        Wait for $U(k,k)$
        Factor the block column $L(k)$
        Send $L(k)$ blocks to needed processes in $P_r(:)$
    else
        Receive $L(k)$ blocks if needed
    if $p_{id} \in P_r(k)$ **then**
        Wait for $L(k,k)$
        Compute the block row $U(k)$
        Send $U(k)$ blocks to required processes in $P_c(:)$
    else
        Receive $U(k)$ blocks if required
Schur-complement update
    if $L(:,k)$ and $U(k,:)$ are locally non-empty **then**
        for $j = k{+}1,\ k{+}2,\ k{+}3\ldots\ n_s$ **do**
            for $i = k{+}1,\ k{+}2,\ k{+}3\ldots\ n_s$ **do**
                if $p_{id} \in P_r(i) \cap P_c(j)$ **then**
                    $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$

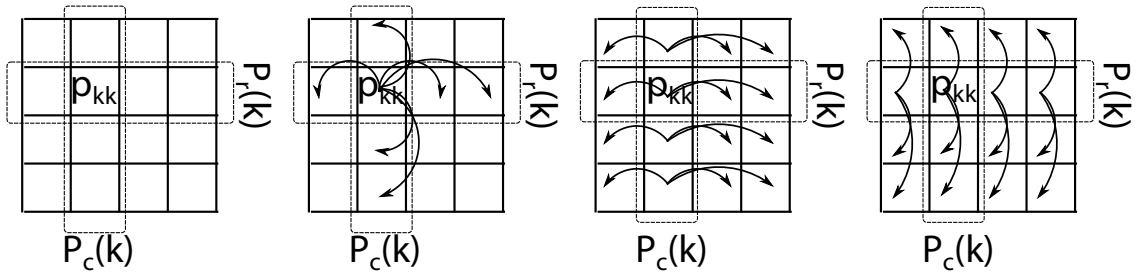


Figure 3.6: We show various MPI communicator and communication pattern involved in the factorization of k-th supernode. In Figure 3.6a, we show the k-th process row $P_r(k)$and process column $P_c(k)$. In k-th diagonal factorization, process p_{kk} factors the k-th diagonal block and broadcast it across $P_c(k)$ and $P_r(k)$ (Figure 3.6b). In k-th diagonal broadcast, process p_{kk} factors the k-th diagonal block and broadcast it across $P_c(k)$ and $P_r(k)$ (Figure 3.6b). In k-th panel broadcast, each process in $P_r(k)$ broadcast calculated U panel to their process column (Figure 3.6c) , and similarily each process in $P_c(k)$ broadcast the calculated L panel to their process row (Figure 3.6d).

- **Diagonal Factorization:** Process p_{kk}, who owns the k-th diagonal block $A(k,k)$, factors it into $A(k,k) = L(k,k)U(k,k)$.[2]
- **Diagonal Broadcast:** Process p_{kk} broadcasts the factored diagonal block $L(k,k)$ across k-th process row $P_r(k)$ and broadcasts $U(k,k)$ across k-th process column $P_c(k)$ (Figure 3.6b).
- **Panel Solve:** Upon receiving $L(k,k)$, processes in $P_r(k)$ calculate $U(k,j) = L^{-1}(k,k)A(k,j)$ for $j > k$. Similarly, on receiving $U(k,k)$, processes in $P_c(k)$ calculate $L(j,k) = A(j,k)U^{-1}(k,k)$ for $j > k$.
- **Panel Broadcast:** After the panel-solve step, processes in $P_r(k)$ broadcast the k-th U panel $U(k,k+1:n_s)$ to respective process columns (Figure 3.6c). Here, n_s is number of supernodes. Similarly, processes in $P_c(k)$ broadcast the k-th L panel $L(k+1:n_s,k)$ to respective process rows(Figure 3.6d).

- ***Schur-complement update:***

[2]The L and U factors overwrite the blocks of A.

On receiving k-th L and U panels, a process can update the blocks of $A(k+1:n_s, k+1:n_s)$, known as Schur-complement update, that it owns.

$$A(i,j) = A(i,j) - L(i,k)U(k,j) \ldots i,j > k \tag{3.3}$$

Note that $A(i,j)$, $L(i,k)$ and $U(k,j)$ are stored in a sparse format. Therefore, the update takes place in three steps.

- **Gather:** First, we pack sparse blocks $L(i,k)$ and $U(k,j)$ into a BLAS-compliant format. Denote these packed blocks by $\tilde{L}(i,k)$ and $\tilde{U}(k,j)$. In this form, we can use highly efficient BLAS-3 calls during the update.
- **GEMM:** We call a (presumably) optimized GEMM to compute the product $V = \tilde{L}(i,k)\tilde{U}(k,j)$, where V is a dense and packed output block.
- **Scatter:** Finally, using the block V, we perform an elementwise update on the $A(i,j)$ block, known as a *scatter* computation. Scatter operations can be, in many cases, as expensive as GEMM operations because they are intrinsically memory bound.

The gather, GEMM, and SCATTER kernels together constitute the Schur-complement update. This is typically the most expensive sub-step of the sparse LU factorization phase.

3.8.3 Elimination tree parallelism

Due to the sparsity of the input matrix, multiple panels can be factorized in parallel. Dependencies between the factorization of different panels is described using the etree. For instance, Figure 3.7a shows a sparse matrix and the corresponding etree in Figure 3.7b. When we factor panel 1 of the sparse matrix, in the Schur-complement update we update the panels. On the other hand, we need not update panels 2, 3

Table 3.1: List of symbols used

Symbol type	Symbol	Def
Process	P	#MPI processes
	P_x, P_y, P_z	Process grid dimensions
	P_{xy}	$P_x \times P_y$ # processes in xy plane
	$P_x(k)$	($k \mod P_x$)-th process row
Matrix & Indexing	A, L, U	The input matrix A and LU factors
	b	Right hand side
Graphs	E	Elimination tree of A
	E_f	Elimination tree-forest (Section 6.3.3)
	S	Top level separator of E
	C_1, C_2	Children etrees of E
Misc.	n	Dimension of the matrix A
	$nlevel$	Height of E $\mathcal{O}(\log n)$
	l	$\log_2 P_z$
	M	Per-process memory
	W	Per-process communication
	L	Latency of factorization
	$T(v)$	*Cost* of factoring node$v$

and 4. Therefore, the factorizations of panels 1, 2, 3 and 4 are independent, and may therefore be performed in parallel.

Factorization of different supernodes can be performed in any order that follows the elimination tree without incurring any additional fill-in.

3.8.4 Lookahead Factorization

SUPERLU_DIST uses etree parallelism to compute the panel-factorization of multiple supernodes in advance. Specifically, before updating the k-th supernode's Schur-complement update, SUPERLU_DIST tries to factor the panels of $k+1,\ldots,k+n_w$ supernodes, where n_w is called the *lookahead window size*. Performing the panel-factorization in advance decreases process idle time spent waiting for L and U panels.

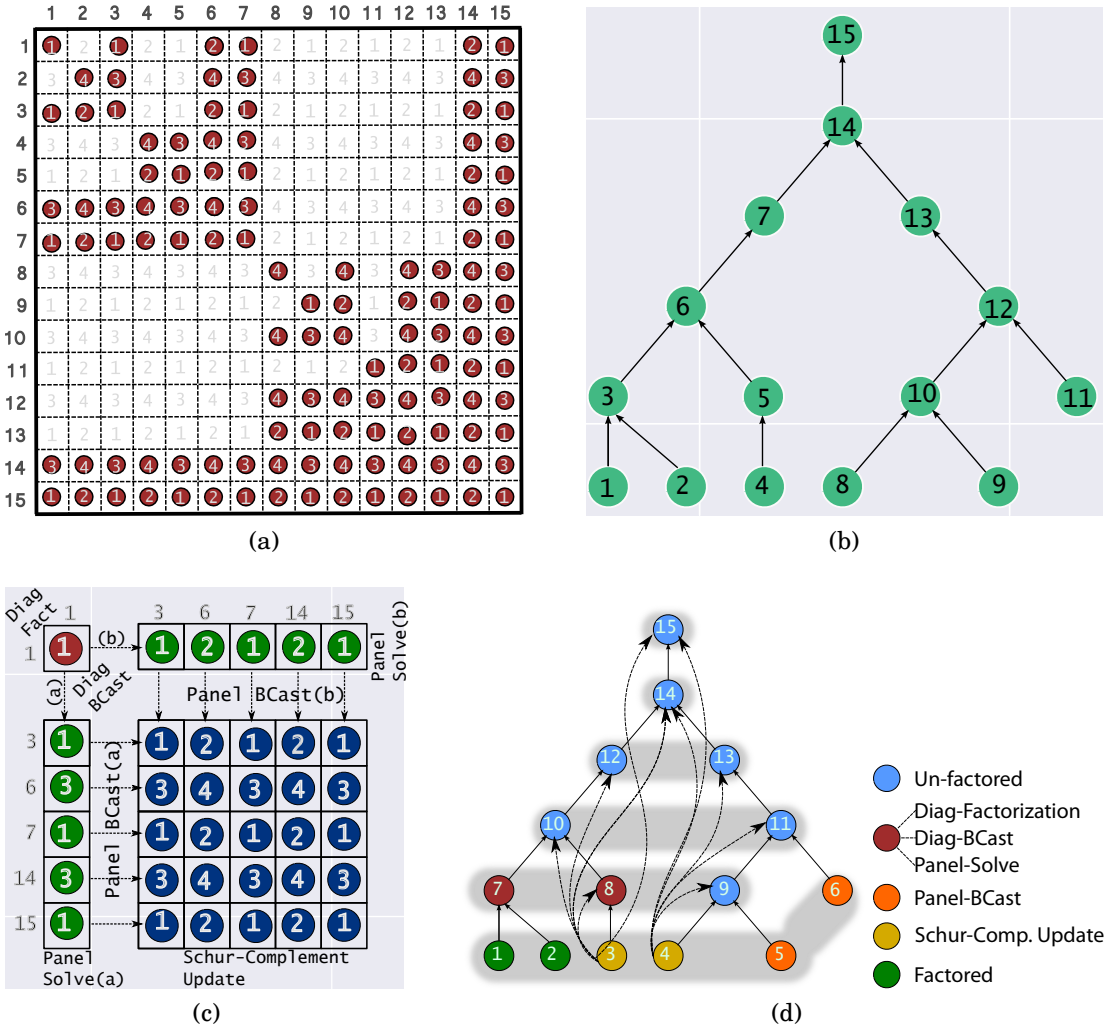


Figure 3.7: A distributed sparse matrix and its etree. Suppose the block sparse matrix of Figure 3.7a is distributed onto a 2×2 process grid. Each circle represents a non-zero block, where the circle's number denotes the process-id that owns the block. Figure 3.7b shows the etree. Figure 3.7c shows the kernels and data involved in factoring supernode 1. Figure 3.7d shows how SUPERLU_DIST uses the etree for pipelining the panel-factorization and Schur-complement update.

3.8.5 Task scheduling and the elimination tree

SUPERLU_DIST uses the etree's parallelism to overlap computation and communication. It concurrently performs the Schur-complement update of a supernode and the panel-factorization of nodes in a so-called *lookahead window* [11]. In the bottom-up ordering of factoring the etree, leaf nodes are factored first. So, panel-factorization of the next several nodes does not depend on the panel-factorization or Schur-complement update of the current node. As such, SUPERLU_DIST performs panel-factorization of the supernodes ahead of their Schur-complement update. But the Schur-complement update of the nodes in the lookahead window cannot be performed in parallel, because the Schur-complements of the leaves may share common blocks of the matrix A. Therefore, SUPERLU_DIST performs the Schur-complement update sequentially for each supernode.

Usually, a large lookahead window creates too many in-flight messages and requires too much buffer space for the incoming messages. So the lookahead window typically has a fixed size in the range 8-20 steps.

CHAPTER 4

A DISTRIBUTED CPU-GPU SPARSE DIRECT SOLVER

4.1 Introduction

Current and emerging systems are relying on manycore co-processors or "accelerators," like GPUs, for improved performance at lower power. In the context of sparse LU, a natural question is how to exploit all forms of available parallelism, whether distributed memory, shared memory, or accelerated.

The challenge is that sparse LU factorization is, computationally, neither strictly dominated by arithmetic, like high-performance LINPACK is when A is dense, nor is it strictly dominated by communication, as is often the case with iterative linear solvers. Thus, it is an open question whether or by how much we should expect to speed up sparse LU factorization using distributed CPU+GPU machines [1]. Additionally, the facts of indirect irregular memory access, irregular parallelism, and a strong dependence on the input matrix's structure—known only at runtime—further complicate its implementation. These complications require carefully designed data structures and dynamic approaches to scheduling and load balancing. Indeed, perhaps due to these myriad issues, there are many studies offering distributed algorithms and hybrid single-node CPU+GPU implementations but, to date, no fully distributed hybrid CPU+GPU sparse direct solver of which we are aware (Section 4.2).

This chapter presents the first such algorithm and implementation that can run scalably on a cluster comprising hybrid CPU+GPU nodes.[1] We extend an existing distributed memory sparse direct solver, SUPERLU_DIST [13], by adding CPU multithreading and GPU acceleration during the LU factorization step. To effectively

[1] This work has been published by us [12].

exploit intranode CPU and GPU parallelism, we use a variety of techniques (Section 4.3). These include aggregating small computations to increase the amount of compute-bound work; asynchronously assigning compute-bound work to the GPU and memory-bound work to the CPU, thereby minimizing CPU-GPU communication and improving system utilization; and careful scheduling to hide various long-latency operations. We evaluate this implementation on two GPU clusters and test problems derived from applications (Section 4.4). We show speedups of up to 3× (Section 4.4, Figure 4.4) over a highly scalable MPI-only baseline; and, when our approach does not yield speedups, explain why.

Beyond the community of users specifically intersted in sparse direct solvers, there may be broader lessons for researchers working at all levels of the stack, including algorithms, programming models, run-time systems, and architectures. Perhaps sparse LU may serve as an important "community benchmark" around which we can improve the overall design of next-generation high-performance software and hardware systems, as we suggest in Section 4.5.

4.2 Related Work

There are a number of successful uses of GPU accelerators to speed up dense linear algebra [14], which naturally motivates questions about how to extend these ideas to the sparse case. One may ask such questions of both sparse direct and sparse iterative methods; given the focus of the present chapter, our survey below in turn focuses on related work in the area of sparse direct methods.

The last five years has seen several research developments on accelerating sparse factorization algorithms using GPUs. Most of these efforts rely on the GPU for solving large dense matrix subproblems, performing any other processing on the host CPU with data transfer as needed. For example, Krawezik and Poole [15], Yu et al. [16], and Vuduc et al. [1] describe such approaches for multifrontal Cholesky

factorization methods. (These approaches transfer each frontal matrix to the GPU for dense operations, and perform assembly operations on the CPU.) Schenk et al. study the left-looking sparse LU factorization in the PARDISO library using single CPU/GPU combination, with single precision [17]. They offload SGEMM, STRSM and SGETRF (dense LU) routines to the GPU when the flop count exceeds an empirically determined threshold (7×10^6 flops) and keep other operations on the CPU. In essence, all of these methods use the GPU as a BLAS accelerator. They consistently achieve several fold speedups over a tuned but single-core CPU code. BLAS acceleration is certainly a sensible design, since only the dense operations have sufficiently high arithmetic intensity to achieve performance gains from GPUs. The assembly procedure involves indirect addressing and scattering operations, and is harder to map efficiently onto GPUs.

George et al. go beyond BLAS acceleration for their single-node multifrontal sparse Cholesky algorithm, implemented in WSMP [18]. They examine three compute-intensive kernels associated with each frontal matrix: factoring the diagonal block, triangular solution, and Schur complement update. These computations are selectively offloaded to the GPU depending on the workload distribution of the flops, which in turn depends on the input matrix. Their method achieves 10-25× speedups over a single-core.

Lucas et al. also developed a multithreaded CPU/GPU sparse Cholesky algorithm [19]. Their method offloads only large frontal matrices, which represent 90% of the flops. On a multithreaded 8-core CPU + 1-GPU system, they show a 5.9× speedup over a single-core code, but only 1.4× speedup over 8-core CPU-only code with multithreading.

Yeralan et al. developed a sparse multifrontal QR factorization algorithm using one CPU-GPU combination [20]. Since sparse QR has intrinsically higher arithmetic intensity than sparse LU, the pay-off of GPU acceleration should be higher. The

novelty in their approach is to move entire subtrees of the assembly tree to the GPU device, which enables simultaneous factorizations of multifrontal fronts and some extend-add updates on the GPU. Their algorithm achieves up to 11× speedup over the single-core CPU code, and 2.5× speedup over a 24-core CPU code.

Our approach also offloads the most arithmetic-intensive part of the workload to GPUs. However, one distinction of our work is that we aim to more fully exploit the available parallelism of a distributed memory system, namely, distributed memory parallelism via MPI combined with intranode parallelism through multithreading and GPU acceleration. While our implementation is specific to SUPERLU_DIST, we believe techniques discussed in this chapter can be extended to other direct solvers.

4.3 Offloading BLAS calls to the GPU

Our work focuses on speeding up the numerical factorization step (Section 3.6 and Algorithm 2). The panel factorization and row computation phases primarily are concerned with communication. By contrast, the Schur complement update phase (lines 15–19) is the local computation that dominates intranode performance. Thus, it is our main target for optimization.

The Schur complement update has two phases: a *matrix multiply* ("*GEMM*") followed by a local *Scatter*. Section Section 4.3.1 describes the baseline SUPERLU_DIST implementation. In our GPU-accelerated approach, we only offload the GEMM phase onto the GPU, while the multicore CPU concurrently executes the Scatter. Although other schemes are possible, our choice is a natural first heuristic: GEMM, being mostly compute-intensive, is a good GPU offload candidate, whereas Scatter, being mostly memory-intensive, is more likely to be limited by PCIe transfer.

Even with this scheme, there are critical details. The computation is sparse, which means the GEMM phase includes many small subproblems that may have relatively low intensity. Therefore, to increase the overall intensity of the GEMM

phase, we explicitly *aggregate* smaller GEMM subproblems into larger ones (Section 4.3.2). Furthermore, we use careful scheduling to hide PCIe transfer costs and overlap GEMM and Scatter phases (Section 4.3.3). Lastly, we parallelize the Scatter phase using OpenMP, with a simple model-driven approach to scheduling (Section 4.3.4). This scheduling "trick" is needed because the cost of each of the GEMM and Scatter phases can vary significantly as the iteration count k increases.

4.3.1 Baseline Schur complement update

The Schur complement update step at iteration k of Algorithm 2 computes $A_{k+1,n_s:k+1,n_s}$ as

$$A_{k+1,n_s:k+1,n_s} = A_{k+1,n_s:k+1,n_s} - L_{k+1:n_s,k} U_{k,:k+1:n_s}. \tag{4.1}$$

SUPERLU_DIST uses an owner-computes strategy, where each process updates the set of blocks, $\{A_{i,j}\}$, which it owns once it has received the required blocks $L_{:,k}$ and $U_{k,:}$.

Each GEMM subproblem computes one $A_{i,j}$, which is line 19 of Algorithm 2. In the baseline SUPERLU_DIST implementation, a process updates each of its $A_{i,j}$ blocks in turn, traversing the matrix in a columnwise manner (outermost j-loop at line 18 of Algorithm 2). The update takes place in three steps: packing the U block, calling BLAS GEMM, and unpacking the result. We refer to the first two steps as the *GEMM* phase, and the last step as the *Scatter* phase.

Packing allows the computation to use a highly optimized BLAS implementation of GEMM. Packing converts the $U_{k,j}$, which is stored in a sparse format, into a dense BLAS-compliant column major format, $\tilde{U}_{k,j}$. This packing takes place once for each $U_{k,j}$. The $L_{i,k}$ operand need not be packed, as it is already stored in a column major form as part of a rectangular supernode.

The second step is the BLAS GEMM call, which computes $V \leftarrow L_{i,k}\tilde{U}_{k,j}$, where V is a temporary buffer.

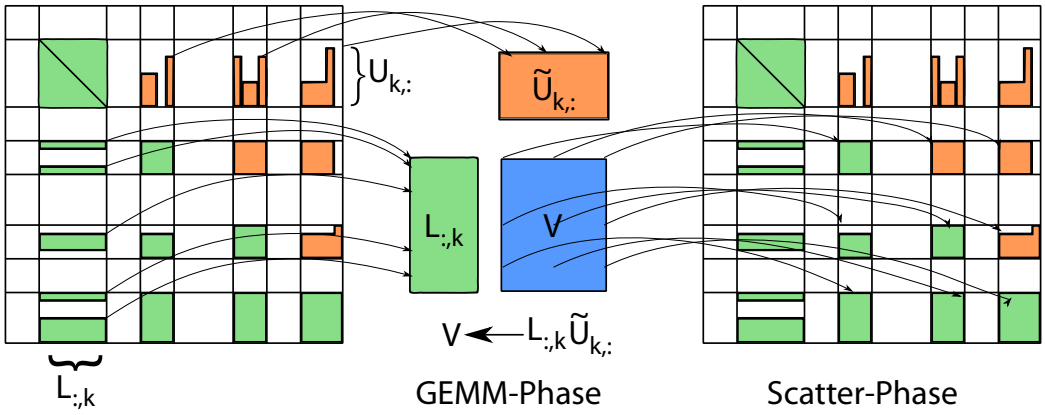


Figure 4.1: Aggregating Small GEMM subproblems

The final *Scatter* step updates $A_{i,j}$ by subtracting V from it. Since only the nonzero rows of L and U are stored, the destination block $A_{i,j}$ usually has more nonzero rows and columns, than $L_{i,k}$ and $U_{k,j}$. Thus, this step must also map the rows and columns of V to the rows and columns of $A_{i,j}$ before the elements of $A_{i,j}$ can be updated, which involves indirect addressing. This final unpacking step is what we refer to as the *Scatter* phase.

4.3.2 Aggregating small GEMM subproblems

Relative to the baseline (above), we may increase the intensity of the GEMM phase by aggregating small GEMM subproblems into a single, larger GEMM. This aggregated computation then becomes a better target for GPU offload, though it also works well even in the multicore CPU-only case.

Our approach to aggregation, illustrated in Figure 4.1 and sketched in Algorithm 3, has two aspects. First, we process an entire block column at once. That is, instead of calling GEMM for every block multiply $L_{i,k}\tilde{U}_{k,j}$, we aggregate the L-blocks in column k into a single GEMM call that effectively computes $V \leftarrow L_{k+1:n_s,k}\tilde{U}_{k,j}$, thereby reusing $\tilde{U}_{k,j}$. Secondly, the packed block $\tilde{U}_{k,j}$ may still have only a few nonzero columns. Thus, we group multiple consecutive U-blocks to form a larger $\tilde{U}_{k,j_{st}:j_{end}}$ block, where j_{st} and j_{end} are the starting and the ending block indices.

Algorithm 3 Schur Complement Update : Aggregating GEMM calls

$r \leftarrow FullRows(L_{:,k})$
$s \leftarrow SuperSize(k)$ ▷ Size of k^{th} Supernode
$MaxCol \leftarrow \min(\frac{\Gamma_1}{r}, \frac{\Gamma_2}{s})$
while $j_{st} < n_s$ **do**
Gather
    $n_{col} \leftarrow 0$
    for $j = j_{st}, j_{st+1}...$ **do**
        $n_p = NumCol(U_{k,j})$
        if $n_{col} + n_p < MaxCol$ **then**
            $\tilde{U}_{k,j} \leftarrow Full(U_{k,j})$ ▷ Storing in dense format
            $\tilde{U}_{:,n_{col}:n_{col}+n_p} \leftarrow \tilde{U}_{k,j}$
            $n_{col} \leftarrow n_{col} + n_p$
        else
            $j_{end} \leftarrow j$
            Break
GEMM
    $V \leftarrow L_k \tilde{U}_{:,1:n_{col}}$ ▷ BLAS Call
Scatter
    $Scatter(A_{k+1:n_s,j_{st}:j_{end}} \leftarrow V_{j_{st},j_{end}})$
    $j_{st} \leftarrow j_{end}$

This large block has some minimum number of columns N_b, a tuning parameter. We schedule the computation of $L_{k+1:n_s,k}\tilde{U}_{k,j_{st}:j_{end}}$ onto the GPU, using CUDA streams as explained below.

Aggregation may increase the memory footprint relative to the baseline. In particular, we may need to store a large U-block, $\tilde{U}$, and a large intermediate output, V. Our implementation preallocates these buffers, using N_b as a tunable parameter to constrain their sizes.

4.3.3 Pipelined execution

Given aggregated GEMMs, we use a software pipelining scheduling scheme to overlap copying the GEMM operands to the GPU with execution of both the GEMMs themselves as well as the CPU Scatter.

Our pipelining scheme, illustrated in Figure 4.2, uses CUDA's streams facility. Our scheme divides $\tilde{U}$ into n_s partitions, where n_s is the number of desired CUDA

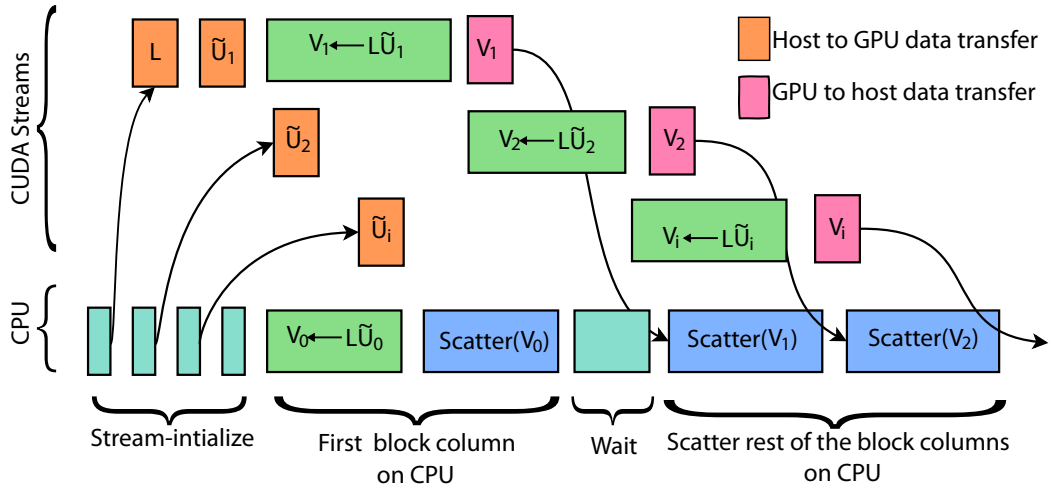


Figure 4.2: Overlapping GEMM with Scatter

streams, a tuning parameter. To perform this division, our scheme first ensures that each partition has a minimum of N_b columns. It also ensures that the number of columns in each partition does not cross the boundary of the block columns. It then uses a greedy algorithm to ensure that each partition has a number of columns of at most the average number of columns, except for the last partition which has all the remaining columns.

The pipelining begins with the transfer of L to the GPU. Now each CUDA stream asynchronously initializes transfer of i-th partition, $\tilde{U}_i$, and a CUDA BLAS GEMM call to perform $V_i \leftarrow L\tilde{U}_i$, and transfer of V_i to the host. Once V_i is copied back to the host, this V_i is scattered as soon as possible. We schedule the GEMM and scatter of the first block column on CPU. This is done to minimize idle time of CPU while it waits for the first CUDA stream to finish transferring the V_1. Note that CUDA streams mainly facilitates overlap of CPU, GPU, and PCIe transfer. The streams themselves may, but do not necessarily, overlap Algorithm 4 describes pseudocode for the pipelined scheduling of DGEMM and Scatter operations.

CUDA streams facility carries a nontrivial setup overhead. Suppose asynchronous CUDA calls take time t_s to initialize, and the effective floating-point throughput of the CPU is F_{cpu} operations per unit time. Then, offloading fewer than $t_s F_{CPU}$ would

Algorithm 4 Pipelined GEMM-SCATTER Scheduling

Require: V, L, $\tilde{U}$,m, n, k,j_{st},j_{end}, n_s, N_b,Fp_{min}

1: Assign first block column j_{st} to CPU
2: $C_{min} = \max(N_b, Fp_{min}/(2mk), N_{col}/n_s)$
3: Partition $j_{st}+1 : j_{end}$ into n_s blocks with each block consists of atleast C_{min} column.
4: AsyncMemcpy(HostToDevice,L,L_d)
Initialization
5: **for** $i = 1 : n_s$ **do**
6: Calculate start st_i and end end_i block $\tilde{U}$ indices for i^{th}
7: AsyncMemcpy HostToDevice
8: $\tilde{U}_{:,st_i:end_i} \rightarrow (\tilde{U}_d)_{:,st_i:end_i}$
9: Async cuBLAS-GEMM call
10: $(V_d)_{:,st_i:end_i} \leftarrow L_d(U_d)_{:,st_i:end_i}$
11: AsyncMemcpy DeviceToHost
12: $(V_d)_{:,st_i:end_i} \rightarrow V_{:,st_i:end_i}$
Start the pipeline by handling first block column on CPU
13: CPU-GEMM $V(:,1) = L_{:,k}\tilde{U}_{:,j_{st}}$
14: CPU-Scatter $Scatter(A_{k+1:n_s,j_{st}} \leftarrow V_{:,j_{st}})$
15: **for** $i = 1 : n_s$ **do**
16: CUDAStreamSynchronize(i)
17: Calculate Start st_i and end end_i block $\tilde{U}$ indices assigned for this stream
18: Scatter $A_{k+1:n_s,st_i:end_i} \leftarrow V_{:st_i:end_i}$

be slower than executing on the host. Our implementation uses such a heuristic to decide whether offloading a particular GEMM phase to the GPU will pay off, or otherwise executes on the CPU.

4.3.4 OpenMP parallelization of Scatter

We parallelized Scatter using OpenMP. There are a number of ways to assign blocks to be scattered to threads. Prior work on SUPERLU_DIST used a block cyclic assignment [11]. However, we discovered by experiment that particular static assignment can lead to severe load imbalance. In addition, assigning one block to a thread can be inefficient since many blocks may have very little work in each, leading to an overly fine grain size.

We address these issues as follows. When there are a sufficient number of block columns, we schedule the Scatter of the entire block column to one thread using

OpenMP's guided scheduling option. We also tried dynamic scheduling options, but for our test cases, there was no significant difference in performance. When there are fewer block columns than the number of threads, we switch from parallelizing across block columns to parallelizing across block rows.

In addition, we also use OpenMP to parallelize the local work at the lookahead phase and the panel factorization phase. However, doing so does not affect performance by much because these phases are dominated by MPI communication.

4.4 Results

Our experiments aim to answer three high-level questions about exploiting intranode parallelism in a distributed memory sparse direct solver based on SUPERLU_DIST.

First, by how much can *explicit* intranode optimization techniques improve performance above and beyond having a highly tuned multicore and/or GPU-accelerated BLAS? (Section 4.4.2) Implicitly, this question is one of productivity, quantifying the profit possible at some level of development cost.

The second question is to what extent do these techniques affect strong scalability? (Section 4.4.3) This question will become more important if the degree of parallelism available in a node increases over time, as most industry observers expect.

The third question is of specific interest to SUPERLU_DIST and sparse direct solvers more broadly: how do time and *storage* trade-off as intranode parallelism increases? (Section 4.4.5) A fact about SUPERLU_DIST is that even if time does not improve with the number of intranode threads, memory requirements can actually decrease. This question is a critical one to applications that use sparse direct solvers. We quantify this effect on our implementation.

Table 4.1: Evaluation testbeds for our experiments

Parameter	Jinx-Cluster	Dirac-GPU test bed
# GPUs per node	2	1
Type of GPU	Tesla M2090 "Fermi"	Nvidia C2050 "Fermi"
GPU double-precision peak	665 GF/sec	515 GF/sec
GPU DRAM / Bandwidth	6 GB / 177 GBytes/sec	3 GB / 144 GBytes/sec
Host	Intel Xeon X5650 @2.67 GHz	Intel Xeon X5550 @2.67 GHz
PCIe / Bandwidth	PCIe x16 /8GB/s	PCIe x16 /8GB/s
Sockets × Cores / socket	2×6	2×4
CPU double-precision peak	128 GF/sec	85 GF/sec
L3 Cache	2 × 12M	2× 8M
Memory / Bandwidth	24GB/42.56 GB/sec	24GB/51.2 GB/sec
Network /Bandwidth	InfiniBand/ 40 Gbit/s	InfiniBand/ 32 Gbit/s

4.4.1 Platforms, matrices, and implementations

We used two GPU clusters in our evaluation (Table 4.2). We tested our implementations on the input matrices in Table 4.2, which derive from real applications [21].

We evaluated 6 implementation variants. (All variants use double-precision arithmetic, including on the GPU.) The baseline is SUPERLU_DIST. We *modified* this baseline to include the BLAS aggregation technique of Section 4.3.2 (Algorithm 3). Since all variants derive from SUPERLU_DIST, they all *include* distributed memory parallelism via MPI. Their mnemonic names describe what each variant adds to the MPI-enabled baseline. • **MKL**$_1$ is the baseline, based on SUPERLU_DIST Version 3.3 "out-of-the-box." It uses MPI-only within a node and uses Intel's MKL, a vendor BLAS library, running in single-threaded mode. This implementation is what we hope to improve by exploiting intranode parallelism. Unless otherwise noted, we try all numbers of MPI processes within a node up to 1 MPI process per physical core, and report the performance of the best configuration. • **MKL**$_p$ is the same as MKL$_1$, but with multithreaded MKL instead. It uses 1 MPI process per socket; within each socket, it uses multithreaded MKL with the number of threads

Table 4.2: Different test problems used for testing solvers. [2]

Name	n	nnz	$\frac{nnz}{n}$	symm	Fill-in Ratio	Application
audikw_1*	943695	77651847	82.28	yes	31.43	structural
bone010*	986703	47851783	48.49	yes	43.52	model reduction
nd24k*	72000	28715634	398.82	yes	22.49	2D/3D
RM07R*	381689	37464962	98.15	no	78.00	computational fluid dynamics
dds.quad†	380698	15844364	41.61	no	20.18	cavity
matrix211†	801378	129413052	161.48	no	9.68	Nuclear Fusion
tdr190k†	1100242	43318292	39.37	no	20.43	Accelerator
Ga19As19H42*	133123	8884839	66.74	yes	182.16	quantum chemistry
TSOPF_RS_b2383_c1*	38120	16171169	424.21	no	3.44	power network
dielFilterV2real*	1157456	48538952	41.93	yes	22.39	electromagnetics

equal to the physical cores per socket. • **{cuBLAS,Scatter}** is $\texttt{MKL}_p$ but with most GEMM calls replaced by their NVIDIA GPU counterpart, via the CUDA BLAS (or "cuBLAS") library. (Any other BLAS call uses $\texttt{MKL}_p$.) Additionally, cuBLAS may execute asynchronously; therefore, there may be an additional performance benefit from partial overlap between cuBLAS and Scatter, as the mnemonic name suggests. Like $\texttt{MKL}_1$, we try various numbers of MPI processes per node and report results for the best configuration. (When there are more MPI processes than physical GPUs, the cuBLAS calls are automatically multiplexed.) • **OpenMP+MKL$_1$** exploits intranode parallelism *explicitly* using OpenMP. It parallelizes all phases using OpenMP. For phases that use the BLAS, we use explicit OpenMP parallelization and with single-threaded MKL. Scatter and GEMM phases run in sequence, i.e., they do not overlap. • **OpenMP+{MKL$_p$,cuBLAS}** shares the work of the GEMM phase between *both* the CPU and GPU, running them concurrently. This tends to reduce the time spent in GEMM compared to $\texttt{OpenMP+MKL}_1$ implementation, but may not hide the cost completely. • **OpenMP+{MKL$_p$,cuBLAS,Scatter}+pipeline** adds pipelining to $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS\}}$. We use $n_s = 16$ CUDA streams and $N_b = 128$.

The first three implementations use implicit parallelism via multithreaded or GPU-accelerated BLAS; the last three involve explicit parallelism. In all cases, we use other default settings for SUPERLU_DIST: MC64 for static pivoting and equilibration, and Metis's serial nested dissection to preserve sparsity. We used $X_s = 144$ as maximum supernode size. To profile the computation's execution time, we use TAU. When we evaluate memory usage (Section 4.4.5), we use the IPM tool [22].

4.4.2 Overall impact of intranode optimization

In this first analysis, we summarize the performance effects of intranode optimization on the two evaluation clusters in Figures 4.3 and 4.4. These experiments use just two nodes of the cluster. The results show best-case improvements of up to 3× using our techniques, and highlight scenarios in which our methods may yield a slowdown.

Each subplot of Figures 4.3 and 4.4 shows time (y-axis) versus implementation variant (x-axis) for a given matrix. The time is normalized to the baseline, with actual baseline execution times in the range of 10 to 1,000 seconds (not shown). Each bar breaks down the execution time into its components, which correspond to different phases of SuperLU. The GEMM phase and SCATTER phase are as described in Section 4.3. The SCATTER phase includes any CUDA stream setup and wait time. The "**Other**" phase has three major components: `MPI_Wait`, `MPI_Recv`, and triangular solve. Since this study focuses primarily on the Schur complement update (GEMM +SCATTER), we do not further breakdown **Other** explicitly. (We further breakdown some of this time in the strong scaling experiments of Section 4.4.3.) When phases may overlap, the bar shows only the *visible* execution time, i.e., the part of the execution time that does *not* overlap. Thus, the total height of the bar is the visible wall-clock time. Lastly, for ease of presentation, each bar is labeled by its speedup relative to the baseline, which is always the leftmost bar ($\texttt{MKL}_1$) in each subplot.

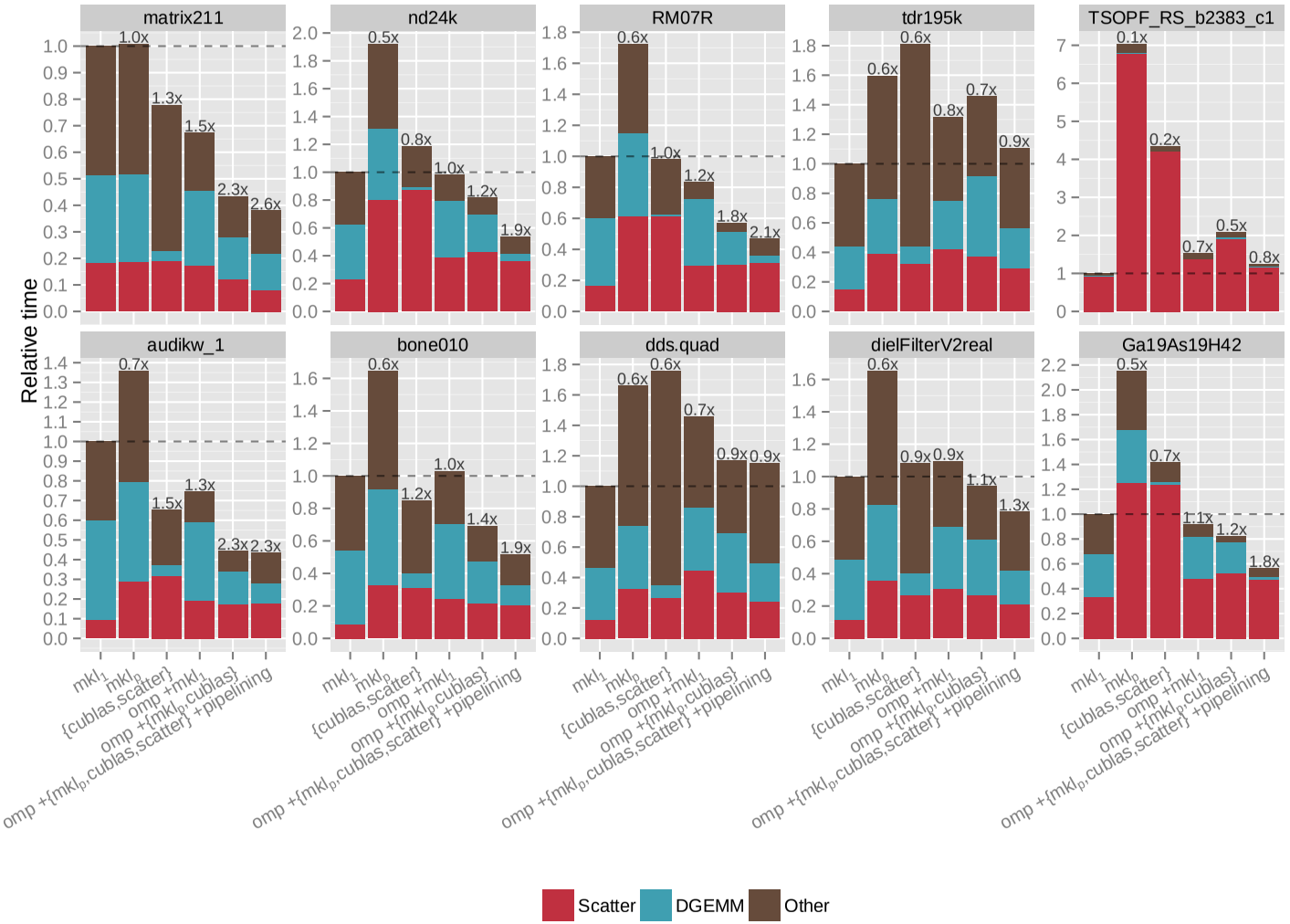


Figure 4.3: Performance comparision of different implementation for different test problems on Dirac Cluster. Each bar is labeled by its speedup relative to the baseline ($\texttt{MKL}_1$).

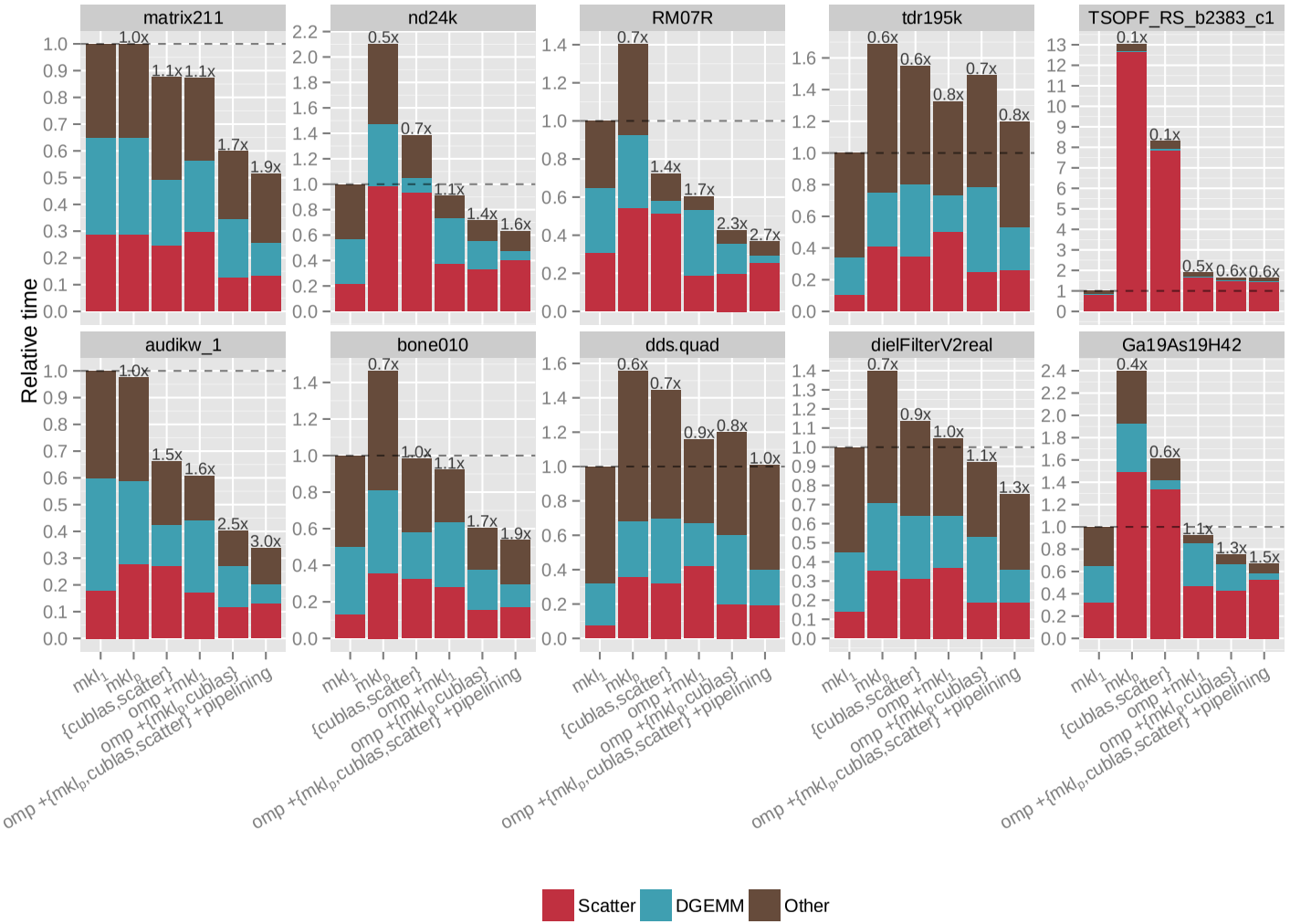


Figure 4.4: Performance of different implementations for different test problems on Jinx cluster. Each bar is labeled by its speedup relative to the baseline ($\texttt{MKL}_1$).

The $\texttt{MKL}_p$ variant is slower than $\texttt{MKL}_1$ in most cases. While the time spent in GEMM remains comparable, SCATTER becomes a bottleneck. Part of the reason is that SCATTER actually scales with the number of MPI processes; since $\texttt{MKL}_p$ uses fewer MPI processes than $\texttt{MKL}_1$, SCATTER time increases. Furthermore, a slow SCATTER phase makes the effect of load imbalance more pronounced, thus increasing the **Other** phase. A similar observation holds for the `{cuBLAS,Scatter}` variant: even when GEMM speeds up compared to $\texttt{MKL}_1$ and $\texttt{MKL}_p$, they tend not to overcome the slowdowns in SCATTER and **Other**. However, due to memory limitations it is not always possible to run $\texttt{MKL}_1$ with one MPI process per physical core. For example, for *matrix211* on Dirac, we could only run 4 MPI processes. In such cases, the $\texttt{MKL}_p$ variant can be faster since it utilizes more physical cores via a multithreaded BLAS. On Jinx, the memory problem is further exacerbated: it has less memory per core, so a larger set of matrices RM07R (20), dielFilterV2real(20), bone010 (12), audikw_1(4) and matrix211 (4) ran on fewer MPI processes than available physical cores. Thus, only relying on accelerating BLAS calls—whether by multithreading or offload—tends not to yield a significant overall speedup, and can in fact decrease performance.

The $\texttt{OpenMP+MKL}_1$ variant reduces the cost of SCATTER and **Other** phases compared to $\texttt{MKL}_p$ and `{cuBLAS,Scatter}`. While **Other** for $\texttt{OpenMP+MKL}_1$ is better than with $\texttt{MKL}_1$, SCATTER is worse. $\texttt{OpenMP+MKL}_1$ often matches the baseline $\texttt{MKL}_1$. The $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS\}}$ variant reduces the time spent in GEMM compared to $\texttt{OpenMP+MKL}_1$ implementation, but cannot hide the cost of GEMM completely. On the Dirac cluster, which has fewer GPUs than CPU sockets, multiple MPI processes end up sharing a GPU, which leads to worse performance. In all cases, the best configuration for $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS\}}$ uses 2 MPI processes with 8 OpenMP threads.

Our combined $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ implementation outperforms $\texttt{MKL}_1$ on 7 of the 10 test matrices on either platform, yielding speedups of

up to 3× (Figure 4.4, *audikw_1*). Compared to $\texttt{MKL}_1$, this variant hides the cost of GEMM very well. However, SCATTER still cannot achieve the same parallel efficiency as with $\texttt{MKL}_1$. Also, observe that accelerating the Schur complement update also reduces the **Other** cost, primarily due to reduced load imbalance. Even when it is slower—matrices *tdr195k*, *TSOPF*, and *dds.quad*—it stays within 10-20% of the MPI-only implementation, i.e., typically showing 0.8-0.9x "speedups." Thus, it can be a reasonable overall candidate when nothing is known about the input problem. The worst case occurs with *TSOPF_RS_bs2383_c1*, which derives from a power network analysis application. On Jinx, it is nearly 2× slower than $\texttt{MKL}_1$ (Figure 4.4). However, even with a slowdown our implementation can reduce the memory requirement of this problem; see Section 4.4.5.

4.4.3 Strong Scaling

Part of the benefit of intranode parallelism is to enhance strong scaling. We consider this scenario, for configurations of up to 8 nodes and 64 cores (Dirac) or 96 cores (Jinx), in Figures 4.5 and 4.6. We present results for just two "extremes": Matrix *nd24k*, on which our implementation does well, and *TSOPF_RS_b2383_c1*, on which it fairs somewhat poorly.

We focus on three of our implementation variants: the baseline $\texttt{MKL}_1$, $\texttt{OpenMP+MKL}_1$, and $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$. For the $\texttt{MKL}_1$ variant, we use 1 MPI process per core. For $\texttt{OpenMP+MKL}_1$ and $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ cases, we use 1 MPI process per socket and one OpenMP thread per core.

Figures 4.5 and 4.6 show scalability as a log-log plot of time (y-axis) versus configuration as measured by the total number of cores (x-axis). Each series shows one of the three implementation variants. Each column is a phase, with the leftmost column, **Total**, showing scalability of the overall computation, inclusive of all phases. Time is always normalized by the *total* $\texttt{MKL}_1$ time when running on the smallest

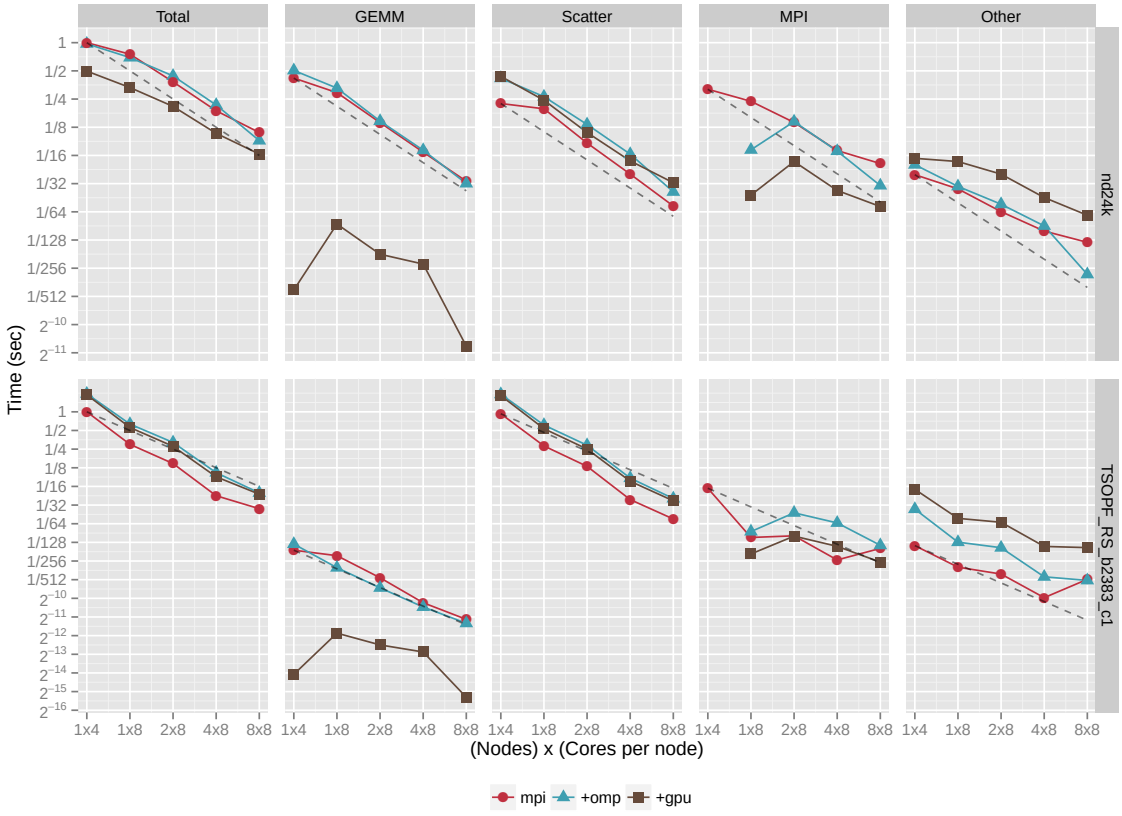


Figure 4.5: Strong scaling on up to 8 nodes (64 cores and 8 GPUs) on Dirac

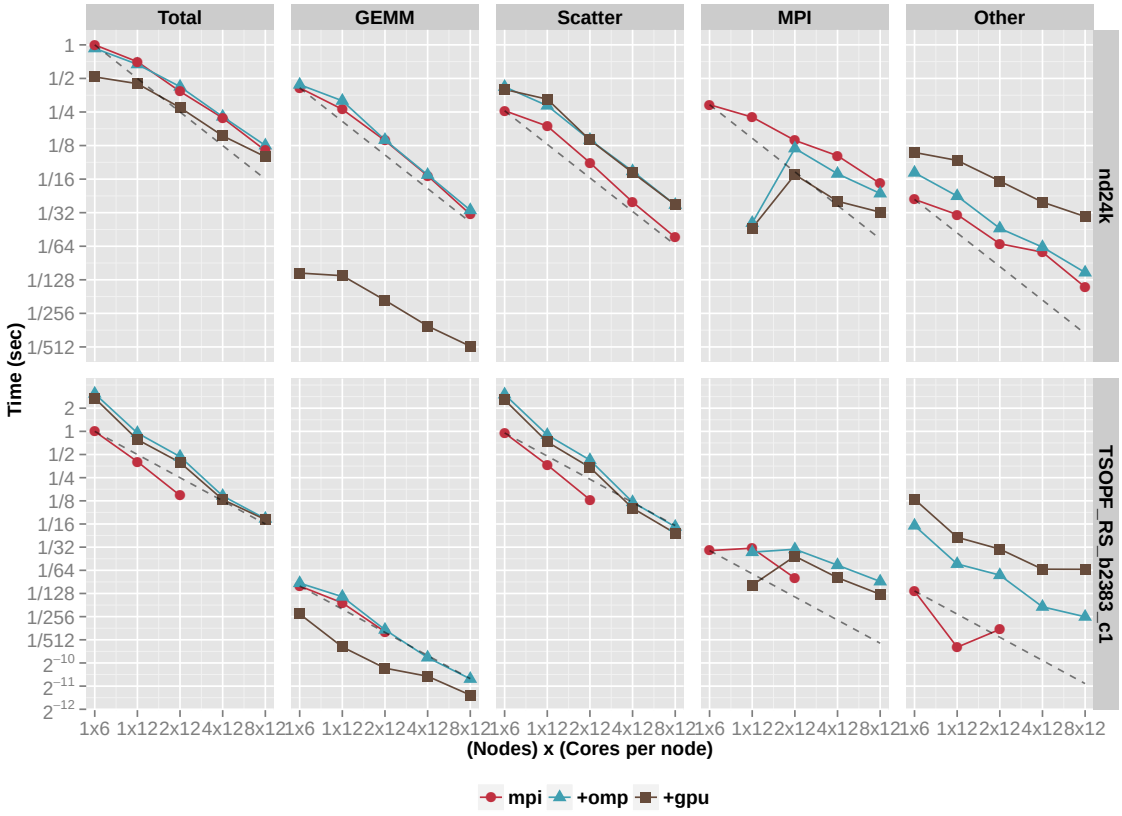


Figure 4.6: Strong scaling on up to 8 nodes (96 cores and 16 GPUs) on Jinx

configuration (1 node and 1 socket), to reveal the relative time spent in each phase. Dashed lines indicate ideal linear speedup for $\texttt{MKL}_1$; perfect scaling would be parallel to this line, while sublinear scaling would have a less steep slope and superlinear scaling would have a more steep slope.

On Dirac (Figure 4.5), both test matrices exhibit good scaling behavior for nearly all the phases. The $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ variant on Dirac is twice as fast for *nd24k* but twice slower for *TSOPF*. But in both cases, scaling improves nearly linearly up to 64 cores and 8 GPUs. (Note that this platform has just 1 GPU per node, so the sudden increase in GPU GEMM time owes to multiplexing of this GPU. This effect does not appear on Jinx, where multiplexing is not necessary.)

On Jinx, scaling is sublinear. At 96 cores and 16 GPUs (2 GPUs per node), all three implementations differ by only a little on *nd24k*. This is due largely to the relatively poor scaling of the **Other** phase, which eventually becomes the bottleneck for $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$.

On *TSOPF_RS_b2383_c1*, the baseline $\texttt{MKL}_1$ is always fastest on both clusters, when it could run. On Jinx, there was not enough memory per node to accommodate the 48 and 96 MPI processes cases, due to the fundamental memory scaling requirement of SUPERLU_DIST; for more analysis, see Section 4.4.5.

For $\texttt{OpenMP+MKL}_1$ and $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ variants in case of 1 node and 4 cores, there is only one MPI process on the node and thus the `MPI_Wait` and `MPI_Recv` costs do not appear. It can be also observed that MPI communication time for $\texttt{OpenMP+\{MKL}_p\texttt{,cuBLAS,Scatter\}+pipeline}$ is twice as fast as $\texttt{OpenMP+MKL}_1$, but with a similar scalability trend.

Matrix *TSOPF_RS_b2383_c1* case shows superlinear scaling. The SCATTER phase is a major contributing factor in cost. As noted previously, the SCATTER phase scales with increasing MPI processes, due primarily to better locality.

Overall, $\texttt{OpenMP+MKL}_1$ shows good strong scaling. By contrast, the scaling of

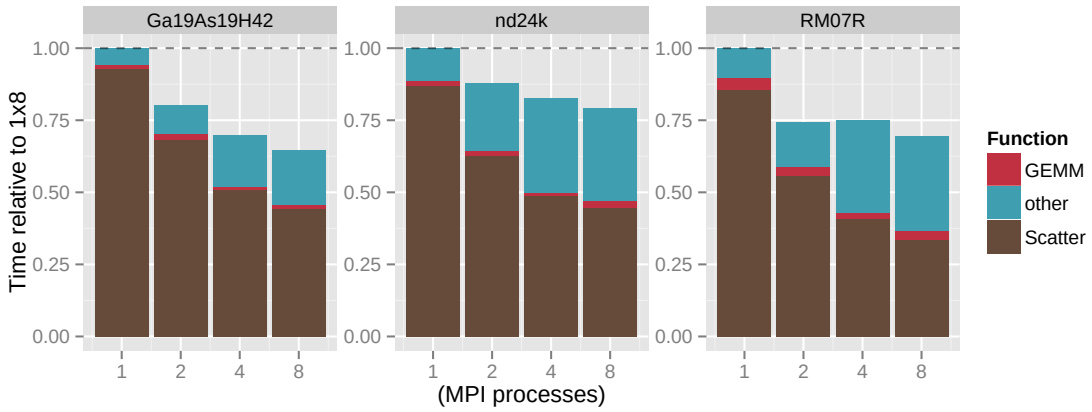


Figure 4.7: Performance on a node equipped with 4 GPUs. Note: In case of 8 MPI processes, 2 MPI process share a GPU

`OpenMP+{MKL`$_p$`,cuBLAS,Scatter}+pipeline` can be worse, as occurs on Jinx. However, this owes largely to Amdahl's Law effects due to **Other**. That component is primarily a triangular solve step, which our work has not yet addressed.

4.4.4 Effect of multiple GPUs on a node

Having multiple GPUs per node allows us to run more MPI processes in a node while reducing or avoiding GPU resource contention. This issue is important because, as observed above, successfully accelerating GEMM means SCATTER can become the bottleneck, and SCATTER scales with the number of MPI tasks.

Our results reflect this benefit. Recall we evaluated two platforms, Dirac with one GPU per node and Jinx with two. We found that having more than two MPI processes sharing a GPU, as on Dirac, can lead to considerable performance degradation due to GPU/PCIe resource contention. By contrast, Jinx suffers less from this limitation. To test this effect directly, Figure 4.7 illustrates the additional performance realizable due to multiple GPUs on a system with 4 C2050 cards and 48 GB of DRAM capacity. We observe that SCATTER shows better scalability with number of MPI processes. However, due to increase in `MPI_Wait` and `MPI_Recv`, the time spent in the **Other** phase also increases. Time spent in GEMM steadily reduces as a fraction of total time

across all the configurations. Overall, having multiple GPUs can improves node level performance.

4.4.5 Time and memory requirements

Sparse direct solvers like SUPERLU_DIST may exhibit a *time-memory tradeoff*. It appears for some especially challenging problems, such as Matrix *TSOPF_RS_b2383_c1*, which was a worst-case example for our approach.

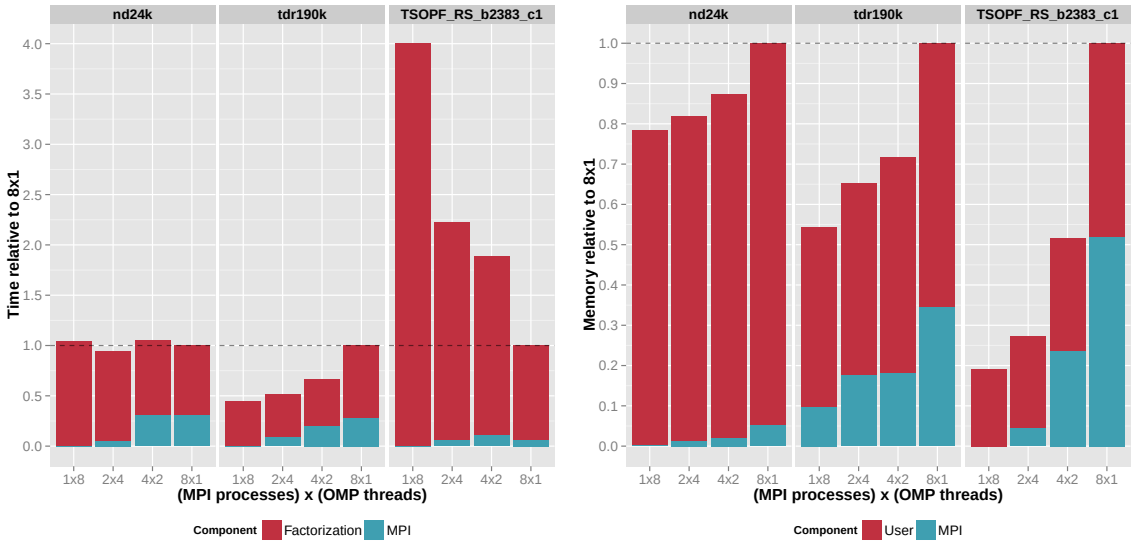
To see the tradeoff, this section considers observations of the $\texttt{OpenMP+MKL}_1$ implementation variant on a single-node of the Dirac system, which has 8 cores per node. When measuring memory, we consider user-allocated memory (by SUPERLU_DIST and our implementation variants) separately from memory allocated by the MPI runtime.

There are three representative problems that exhibit the general range of time-memory behaviors: *nd24k*, *tdr190k*, and *TSOPF_RS_b2383_c1*. We show both time and memory behavior for these three problems in Figure 4.8. In particular, we observe all combinations of (number of MPI processes) × (number of OpenMP threads) = 8. Time and memory are further broken down into the cost due to MPI versus the remaining cost.

Matrix *nd24k* exhibits the general behaviors that we most commonly observed among our test matrices. The best case in execution time occurs for (2 MPI processes) × (4 OpenMP threads), for reasons observed in Section 4.4.2 and Section 4.4.3: as the number of MPI processes increases, the time spent in `MPI_Wait` and `MPI_Recv` tends to increase while the time for Scatter decreases, yielding an optimum.

Matrix *tdr190k* exhibits a different behavior, namely, a strong benefit from intra-node threading. Both time and memory *decrease* with decreasing MPI processes and increasing OpenMP threads.

Matrix *TSOPF_RS_b2383_c1* exhibits the time-memory tradeoff. The SCATTER



(a) Time in MPI vs. compute

(b) User vs. MPI-runtime memory

Figure 4.8: Effect of intranode threading on memory and time

phase dominates execution time, as Section 4.4.3 shows; since SCATTER scales with MPI processes, the all-MPI configuration wins. However, memory usage actually *increases* with increasing numbers of MPI processes. Among user allocated memory, it turns out that the memory required by the L and U factors remains fairly constant, whereas the buffers used for `MPI_Send` and `MPI_Recv` increase. Memory allocated by MPI runtime also increases. Thus, even if our intranode threading approach is slower than the all-MPI case, there can be a large reduction in the memory requirement.

Overall, different phases can contribute different amounts to the total time. This, in turn, decides the relative performance of the of `OpenMP+MKL`$_1$ versus `MKL`$_1$. In a large number of cases, we see that `OpenMP+MKL`$_1$ achieves comparable parallel efficiency as `MKL`$_1$ case. The advantage of `OpenMP+MKL`$_1$ is more apparent on those cases where, due to large memory requirement, it is not possible to use all cores. It is expected that future clusters and accelerators such as Xeon Phi would have more cores and lesser per core memory, thus limiting the use of an MPI-only approach.

4.5 Conclusion

In this chapter, we considered the high-level question that how to exploit intranode parallelism in emerging CPU+GPU systems for distributed memory sparse direct solvers. At the outset, one expects a highly tuned multicore and/or GPU BLAS will yield much of the potential performance benefits. The real question, then, is how much *additional* performance gain is possible from explicit parallelization. Indeed, such questions are being asked about many other kinds of computations, with developers wondering how much effort is necessary to yield a certain return in performance.

Our results for SUPERLU_DIST suggest that on today's systems, there may be up to a factor of 2× more to gain above and beyond BLAS-only parallelization. So, one new question is how to achieve this additional factor.

Another question is how much additional gain might be possible if more of the computation can be offloaded onto the GPU. The Scatter phase is the obvious target, since it could benefit significantly from the relatively high amount of memory bandwidth.

In the following chapter, we consider techniques that can address these questions.

CHAPTER 5

REDUCING INTRA-NODE COMMUNICATION IN HYBRID DIRECT SOLVER

5.1 Introduction

The method of the previous chapter, which enhances SUPERLU_DIST [23] to use GPUs, is limited in scope. It concentrates primarily on how to offload certain dense BLAS subproblems, such as dense GEMM. Such an approach is a natural first step. Indeed, it mirrors much of the existing work, which—until our GPU cluster work—considered only the *single-node* case [24, 25, 26, 27, 28]. However, it also falls short of what is ultimately possible. In this chapter, we address this shortcoming.[1]

For instance, we estimated the best-case speedup of our prior approach on one of the test problems and platforms considered in this study. If GEMM cost *zero* time units, that speedup would be *at most* 1.4×. The method proposed herein can, by contrast, improve that speedup to 1.7× on the same test problem.

By the way of explanation, the idea of offloading BLAS calls is not unreasonable. Recall that the most computationally expensive step in sparse LU factorization is Schur-complement update, which consists of two steps: multiplying two dense matrices (the GEMM step) and scattering the output of GEMM back into sparse format (the SCATTER step). Even if one only offloads large GEMMs, as we did previously so that PCIe transfer costs would not dominate, a non-offloaded SCATTER either becomes a bottleneck or—as multicore CPU memory bandwidth improves—becomes fast enough that overlapping with PCIe transfer for GEMM is no longer effective.

Instead, we propose a new approach based on the high-level idea of using asyn-

[1] We have published the work that appears herein [29].

chronous execution as aggressively as possible. Our algorithm is structurally similar to communication-optimal 2.5D LU factorization, which uses redundancy across processes to reduce network communication [30]. In our case, we use redundancy between the CPU and the co-processor to reduce PCIe communication. However, translating the same high-level idea to sparse LU factorization is much harder than the dense case, due to the irregular parallelism, irregular dependencies, and irregular data structures. We find it necessary to break some of the usual algorithmic abstraction boundaries, fusing distinct steps, such as GEMM and SCATTER, and using asynchrony to do so across iterations. In addition, we combine asynchrony with accelerated offload, lazy updates, and data shadowing (*a la* halo or ghost zones). This combination hides and reduces communication, whether to local memory, across the network, or over PCIe. We refer to this combined technique as the HALO algorithm, where the term HALO evokes *highly asynchronous lazy offload*.

We further enhance the basic HALO framework in two ways, to make it more effective in practice. First, we develop an empirical model-driven autotuning scheme to load balance within the node. This balancing occurs among *both* CPU cores *and* co-processor accelerators. The scheme overcomes limitations of both static load balancing, which can fail to accommodate the intrinsic dynamic and irregular nature of a sparse direct solver; and dynamic load balancing, which may incur high latency overheads due to PCIe. Secondly, we address the memory requirement problem of sparse direct solvers, by implementing a scheme that graceful degrades when offloading to an accelerator whose memory is much smaller than the host's memory. This is done by a heuristic that exploits the structure of a sparse direct solver's *elimination tree*. These enhancements to HALO make it practical.

Although HALO specifically concerns intra-node performance, it is easy to add our single-node implementation into a distributed memory code—namely, SUPERLU_DIST—and thereby accelerate the distributed case. The asynchronous nature of our ap-

proach naturally accommodates overlapping network communication with various on-node tasks. Additionally, although our experimental platform uses MIC co-processors, the technique is generic and could in principle apply to GPU-based platforms. Our hybrid MIC-accelerated SUPERLU_DIST achieves speedups of up to 2.5× on practical problems of interest (Sections 5.7 and 5.8), relative to a highly scalable hybrid MPI+OpenMP baseline. To better understand performance, we analyze our code's performance issues and quantify the *potential* improvements. Together with our scaling experiments, this analysis helps us estimate the potential for future improvements in hardware, software, and runtime systems. Such findings may be of interest beyond the specific case of a sparse direct solver.

5.2 The design space of MIC-based SUPERLU_DIST

In contrast to a GPU, which today may only used as a co-processor for offloading, MIC has two possible execution modes. The first is similar to GPU offloading, and so is referred to as *offload mode*. For SUPERLU_DIST, one could offload compute intensive steps like GEMM to MIC. The second option is to use the MIC as an independent multicore node, launching multiple MPI processes onto MIC to run SUPERLU_DIST; this mode is called *native* mode.

In SUPERLU_DIST, where many calculations are sequential or lack enough parallelism, native mode is not likely to perform well. A single MIC core is slower than a typical high-end CPU core, as it executes in-order, at a lower operating frequency, and with a higher cache miss penalty. On the other hand, if we spawn heterogeneous MPI processes on both the CPUs and the MICs, load balancing among the CPUs and the MICs becomes difficult. Moreover, the MIC has a relatively low memory capacity, which limits the use of the MIC in native mode to matrices of relatively small sizes (Table 5.3), compared to what is possible on the CPU-based host. Thus, our approach focuses on using offload mode.

So what should be offloaded? Recall that there are two main phases, the panel-factorization and the Schur-complement update. Panel-factorization typically has insufficient parallelism for the MIC, and for a small number of MPI processes, it is also not usually the performance bottleneck. At relatively larger numbers of MPI processes, MPI communication costs dominate panel-factorization, which MIC acceleration cannot improve. Therefore, we do not consider this phase for offload.

By contrast, the Schur-complement update phase has a large number of independent GEMM and SCATTER calls that can account for more than 70-80% of the factorization time. Thus, this phase is a good offload candidate.

Our prior approach offloaded the Schur-complement update's GEMM calls to the GPU [31]. In each iteration, it offloaded a large GEMM call that multiplies L and U panel matrices, as $V^{m_t \times n_t} = -L^{m_t \times k_t} U^{k_t \times n_t}$, where typically $m_t, n_t \gg k_t$. Doing so required first sending the L and the U panel matrices to the GPU, then calling CUBLAS to compute GEMM, and then sending the product matrix V back to the CPU via PCIe. The SCATTER of V would occur on the CPU. To hide the data transfer costs, our prior approach pipelined the transfer of V and execution of the SCATTER.

5.2.1 Limitation of BLAS offload

This approach has two critical limitations. First, the bandwidth-bound SCATTER calls, since they remain on the CPU, cannot benefit from high GPU memory bandwidth. In the best test case of this paper, leaving SCATTER unaccelerated on a 20-core Intel *Ivybridge* system yields a maximum possible speedup of 1.4×, even if we assume the GEMM cost to be zero. Secondly, modern CPU bandwidth is significantly higher than the PCIe bandwidth. Therefore, efficiently pipelining PCIe transfer and SCATTER of V is not possible, since the SCATTER time is dwarfed by PCIe transfer time. Thus, our proposed algorithm will try to extend this prior approach by also finding a way to effectively offload SCATTER calls to MIC.

5.3 HALO Algorithm for co-processor offload

To understand our new HALO algorithm, it helps to start with a natural and simpler method. For additional simplicity, first consider the single node case, which we will subsequently extend for the distributed memory case.

5.3.1 A primitive offload algorithm

Recall the k-th iteration of Algorithm 2. It factors the k-th *panel matrices*, $A(k{:}n_s,k)$ and $A(k,k{+}1{:}n_s)$, during the panel-factorization phase, producing the factored panels, $L(k)$ and $U(k)$. In the Schur-complement update phase, it updates the k-th Schur-complement $A(k{+}1{:}n_s,k{+}1{:}n_s)$ by,

$$A(k{+}1{:}n_s,k{+}1{:}n_s) \leftarrow A(k{+}1{:}n_s,k{+}1{:}n_s) - L(k)U(k).$$

Here is a primitive algorithm to offload the Schur-complement update phase to the MIC. This algorithm keeps a copy of the matrix A on the MIC. In the k-th iteration, it transfers the k-th panel matrices from the MIC to the CPU. Using the k-th panels, it calculates the factored panels $L(k)$ and $U(k)$ on the CPU. It then sends the $L(k)$ and $U(k)$ panels to the MIC and updates the k-th Schur-complement on the MIC. Thus, in each iteration this algorithm transfers a pair of panel matrices in each direction; these matrices are considerably smaller than the k-th Schur-complement. Consequently, the PCIe communication volume in each iteration can be relatively small. However, many iterations will not have enough parallelism to utilize the MIC well; therefore, those iterations may be significantly slower on the MIC than on the CPU. To avoid such slowdown, we instead consider *selectively offloading* the Schur-complement update to the MIC.

Algorithm 5 SUPERLU_DIST with MIC-offloading

1: Initialize $A_\phi \leftarrow 0$
2: **for** $k = 1,\ 2,\ 3 \ldots\ n_s$ **do**
3: **Panel Factorization**
4: same as Algorithm 2
5: $\vdots$
6: **Fetch and Assemble matrix on the CPU**
7: (†) the MIC sends $A_\phi(k{+}1{:}n_s, k{+}1)$ and $A_\phi(k{+}1, k{+}1{:}n_s)$ blocks to the CPU
8: **Hybrid Schur-complement Update**
9: **if** $L(k)$ and $U(k)$ are locally non-empty **then**
10: find n_ϕ such that $k < n_\phi \leq n_s$
11: (‡) send $L(k)$ and $U(k, n_\phi : n_s)$ to the MIC
12: **Schur Complement Update on the CPU**
13: **for** $j = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_\phi - 1$ **do**
14: **for** $i = k{+}1,\ k{+}2,\ k{+}3 \ldots\ n_s$ **do**
15: **if** $p_{id} \in P_r(i) \cap P_c(j)$ **then**
16: $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$
17: **Schur-complement Update on the MIC Asynchronously**
18: Wait for (‡) to finish
19: **for** $j = n_\phi,\ n_\phi + 1,\ n_\phi + 2 \ldots\ n_s$ **do**
20: **for** $i = k{+}2,\ k{+}3,\ k{+}4 \ldots\ n_s$ **do**
21: **if** $p_{id} \in P_r(i) \cap P_c(j)$ **then**
22: $A_\phi(i,j) \leftarrow A_\phi(i,j) - L(i,k)U(k,j)$
23: **Reduce the MIC updates with the CPU**
24: the CPU waits for (†) to finish
25: $A(k{+}1{:}n_s, k{+}1) \leftarrow A(k{+}1{:}n_s, k{+}1) + A_\phi(k{+}1{:}n_s, k{+}1)$
26: $A(k{+}1, k{+}2{:}n_s) \leftarrow A(k{+}1, k{+}2{:}n_s) + A_\phi(k{+}1, k{+}2{:}n_s)$

5.3.2 The HALO algorithm

The HALO algorithm enables selective offloading of the Schur-complement update to the MIC by extending the primitive algorithm as shown in Algorithm 5, summarized as follows.

HALO keeps the matrix A on the CPU and keeps a *structural* copy of the matrix A on the MIC, which it initializes with *zeros*. In other words, the matrix on the MIC has the same sparse data structure as the matrix A, but all the stored non-zero entries are initialized to zero. We denote the matrix on the MIC as A_ϕ to distinguish it from the matrix A on the CPU. In the k-th iteration, HALO transfers the k-th panel matrices from the MIC to the CPU. And, on the CPU, it *reduces* the k-th panels from

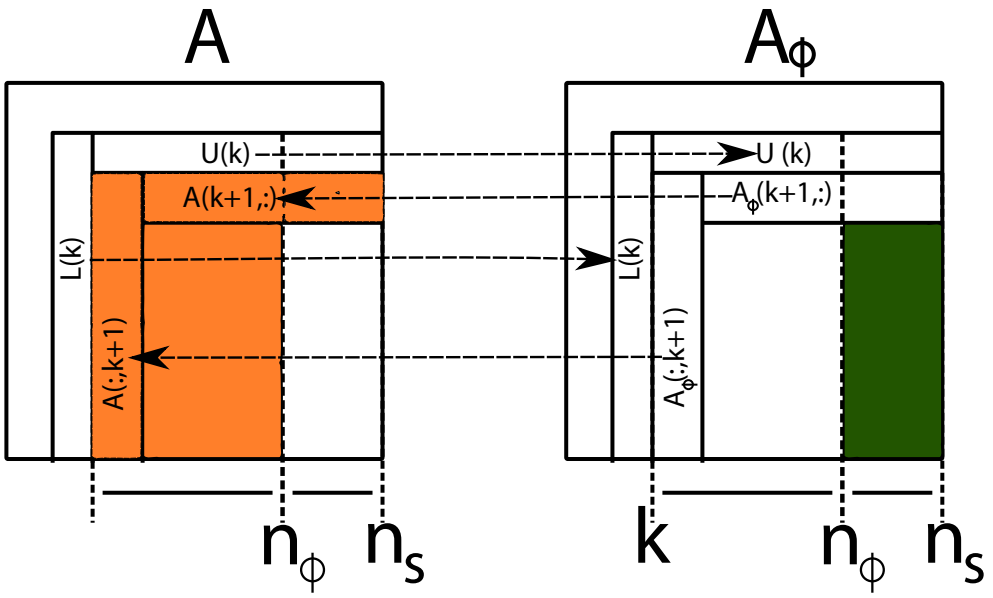


Figure 5.1: HALO: Schur-complement update in the k-th iteration. The $L(k)$ and $U(k)$ panels—calculated in k-th panel-factorization on the CPU—are sent to the MIC. The MIC sends $(k+1)st$ A-panels to the CPU. The CPU and MIC update parts of the k-th Schur-complement, shown in orange for the CPU and in green for MIC. The CPU merges the received MIC's $(k+1)st$ A-panels with its own $(k+1)st$ A-panels, before $(k+1)st$ iteration starts.

the CPU and the MIC, via

$$A(k{:}n_s,k) \leftarrow A(k{:}n_s,k)+A_\phi(k{:}n_s,k); \tag{5.1}$$

$$A(k,k{+}1{:}n_s) \leftarrow A(k,k{+}1{:}n_s)+A_\phi(k,k{+}1{:}n_s). \tag{5.2}$$

then factors the reduced k-th panel matrices, yielding $L(k)$ and $U(k)$ on the CPU. If it offloads the Schur-complement update to the MIC, then it also sends the $L(k)$ and $U(k)$ panels to the MIC and updates the k-th Schur- complement there; otherwise it does the update on the CPU.

In general, HALO divides the k-th iteration's Schur-complement update between the CPU and the MIC. For some value n_ϕ, it updates the submatrix $A(k{+}1{:}n_s,k{+}1{:}n_\phi)$ on the CPU and $A(k{+}1{:}n_s,n_\phi{+}1{:}n_s)$ on the MIC. We discuss how to choose n_ϕ in Section 5.5.

Figure 5.1 illustrates the regions of the matrix updated on and transferred from both the CPU and the MIC.

Note that in the k-th iteration, HALO does not update the $k+1$-th panels on the MIC. This way the MIC can start the transfer of the $k+1$-th panel before it executes the k-th Schur-complement update. Thus, HALO transfers the $k+1$-th panels and updates the k-th Schur-complement on the MIC in parallel. Figure 5.2 shows a sample execution timeline.

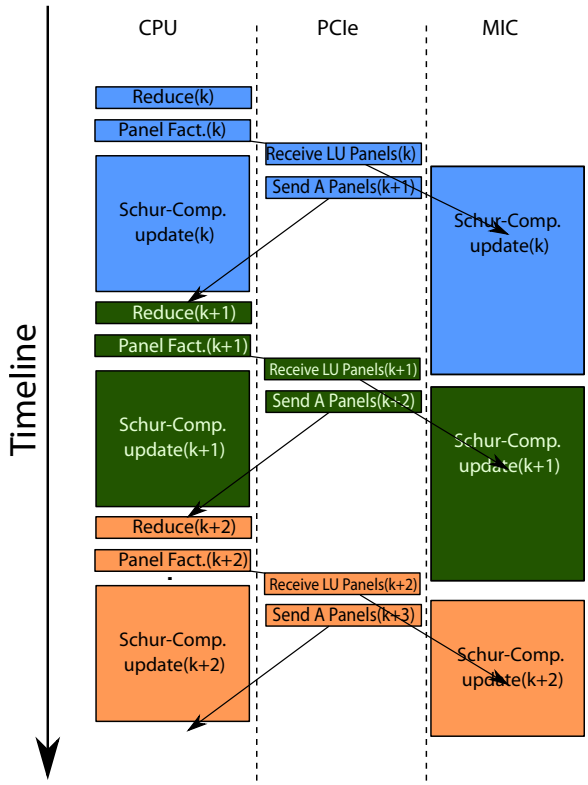


Figure 5.2: Concurrent execution of the Schur-complement update on the CPU and the MIC. *Send A panels(k)* denotes transfers of k-th panels of A_ϕ from MIC to CPU. *Reduce(k)* denotes reductions of k-th panels from the CPU and the MIC. *Receive LU panels(k)* denotes transfers of panels $L(k)$ and $U(k)$ from CPU to MIC. In general, the Schur-complement update is much longer than both other steps and data transfer.

To see how this method works, consider any block $A(i,j)$ and its corresponding $A_\phi(i,j)$ on the MIC. Let $A^0(i,j)$ denote the initial value of $A(i,j)$ and recall that $A_\phi(i,j)$ is initially zero. In some iteration $k < \min(i,j)$, HALO either updates $A_\phi(i,j) \leftarrow A_\phi(i,j) - L(i,k)U(k,j)$ on the MIC or it updates $A(i,j) \leftarrow A(i,j) - L(i,k)U(k,j)$

on the CPU. Let $\mathcal{K}_1$ denote the set of iterations in which $A_\phi(i,j)$ is updated on the MIC, and let $\mathcal{K}_2$ denote those iterations in which $A(i,j)$ is updated on the CPU. Then, the snapshots of $A(i,j)$ and $A_\phi(i,j)$ are given by:

$$A_\phi(i,j) \leftarrow -\sum_{k\in\mathcal{K}_1} L(i,k)U(k,j); \tag{5.3}$$

$$A(i,j) \leftarrow A^0(i,j) - \sum_{k\in\mathcal{K}_2} L(i,k)U(k,j). \tag{5.4}$$

Were we to add $A_\phi(i,j)$ to $A(i,j)$, that would be the same as updating $A(i,j)$ on $\mathcal{K}_1$ iterations, i.e.,

$$\begin{aligned} A(i,j) &\leftarrow A(i,j) + A_\phi(i,j) \\ &= A^0(i,j) - \sum_{k\in\mathcal{K}_1\cup\mathcal{K}_2} L(i,k)U(k,j). \end{aligned}$$

Thus, before the $k = \min(i,j)$-th iteration begins, we can fetch the block $A_\phi(i,j)$ and add it to $A(i,j)$. This reduced $A(i,j)$ block contains updates from all of the $\mathcal{K}_1 \cup \mathcal{K}_2$ iterations. Hence, when participating in the $k = \min(i,j)$-th panel-factorization, the $A(i,j)$ block in the MIC offload case is the same as in non-offloaded case. This argument holds for all the blocks participating in the k-th panel-factorization. Consequently, the factored panels $L(k)$ and $U(k)$ are the same in the case of MIC offload as they would have been otherwise.

Distributed HALO

In SUPERLU_DIST, each process owns a subset of the blocks of A following a 2D-cyclic data distribution. In the distributed HALO, we assign one MIC to each MPI process. Thus, we can conveniently assume that the shadow matrix, A_ϕ, has the same distribution the MICs as A.

Like the single node case, in each iteration's distributed panel-factorization, each process calculates or receives from other processes $L(k)$ and $U(k)$ blocks. It then transfers the $L(k)$ and $U(k)$ blocks to the MIC, and the CPU and the MIC can now update respective Schur-complement in parallel.

In contrast to single node case, for a given k only a subset of processes will own the blocks of $k+1$-th panels, following from the 2D cyclic distribution of the matrix. Therefore, in the k-th iteration, a process only needs to fetch the $k+1$-th panels from the MIC if it owns the $k+1$-th panel blocks, thereby further reducing the overall PCIe transfer volume.

5.4 Intra-node Optimizations

To make HALO practical for wide range of matrices, we augment HALO with several more performance optimizations. These optimizations expose parallelism at all levels of SIMD, multithreading, accelerator, and MPI, and reduces PCIe transfer volume and intra-node synchronizations.

Due to a lack of space, here we describe the two key *algorithmic* enhancements for HALO.

5.5 Model-driven autotuning of intra-node load balance

Intra-node load balance i.e. choosing the optimal value of n_ϕ in Figure 5.1 is vital to achieving good load balance. However, the relative performance of the GEMM and the SCATTER kernel depends strongly on their input operand sizes. This motivates a model-driven scheme for choosing n_ϕ.

By way of motivation, consider Figure 5.3. It shows, for GEMM operations at various problem sizes, the speedup of MIC relative to a dual-socket 10-core Ivy Bridge system. While the theoretical *peak* performance of MIC is twice that of the aggregate peak of the Ivybridge system, Figure 5.3 clarifies that for a wide range of input sizes,

the CPU can be much faster than MIC. As a function of problem size, the relative performance varies widely and nonlinearly.

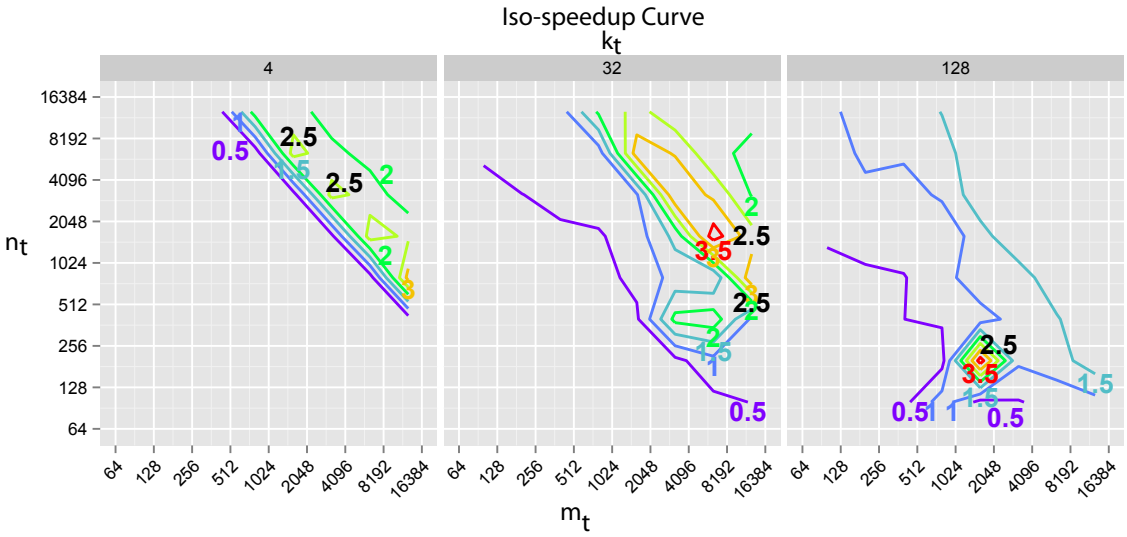


Figure 5.3: The speedup of MIC over a 20-core Ivy Bridge EP server varies widely and nonlinearly, for a GEMM that multiplies a $m_t \times k_t$ by a $k_t \times n_t$ matrix. (Speedups are shown as contour lines.)

Similarly, the performance of SCATTER on MIC also depends on the input block size, as Figure 5.4 shows. Due to MIC's in-order execution, it is crucial to use SIMD and software prefetching; however, for small blocks, it is hard to use SIMD and prefetching effectively. From one iteration to another, the distribution of block sizes in the Schur-complement update varies a lot, and small blocks are common.

MDWIN

Among conventional generic approaches to load balancing, static load balancing is hard to do well under such workload variability, and dynamic load balancing over PCIe may incur high latency overheads. Instead, we propose a model-driven auto-tuning scheme, which we call MDWIN.

MDWIN is a static approach driven by an empirical performance model. At a high level, it tries to predict the execution time of the Schur-complement update using a simple analytical model, whose parameters derive from offline benchmarks. The

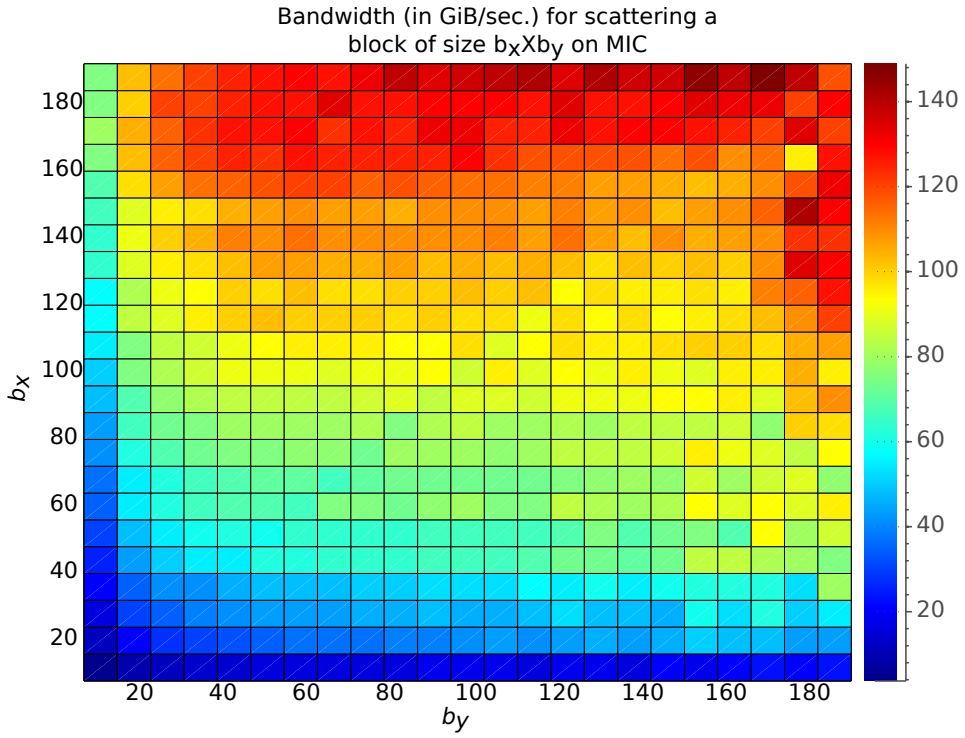


Figure 5.4: When scattering small blocks ($b_x \times b_y$), performance suffers due to poor SIMD and prefetch efficiency.

model is calibrated on both the CPU and MIC, and is used to predict the value of n_ϕ at which the CPU and MIC are approximately balanced in time.

Let t_{GEMM}, t_{SCATTER}, $t^{(\phi)}_{\text{GEMM}}$, $t^{(\phi)}_{\text{SCATTER}}$ be the GEMM and SCATTER times on the CPU and MIC, respectively. In each iteration k, these are functions of n_ϕ. MDWIN seeks n_ϕ such that:

$$t_{\text{GEMM}} + t_{\text{SCATTER}} \approx t^{(\phi)}_{\text{GEMM}} + t^{(\phi)}_{\text{SCATTER}}. \tag{5.5}$$

The component times of this model are determined as follows.

Modeling GEMM *costs*

To determine t_{GEMM} and $t^{(\phi)}_{\text{GEMM}}$, MDWIN maintains a lookup table of flop rates for $V \leftarrow L \cdot U$, where V is $m_t \times n_t$, L is $m_t \times k_t$, and U is $k_t \times n_t$. The sizes may be taken at

a sample of points, the number of which may be used to tradeoff the table size and construction time. Let $F(m_t, n_t, k_t)$ denote this table of flop rates for the CPU. We simply estimate $t_{\text{GEMM}} = 2m_t n_t k_t / F(m_t, n_t, k_t)$. A similar table and formula for $t^{(\phi)}_{\text{GEMM}}$ may be constructed on MIC.

Modeling SCATTER *costs*

SCATTER is a memory bandwidth-bound kernel. Scattering a block of size $b_x \times b_y$ requires $3b_x \times b_y$ memory operations.[2] On the CPU, we observed that in most cases, only a few threads were sufficient to achieve close to *stream bandwidth*, denoted B_{stream}. Therefore, on the CPU, we estimate t_{SCATTER} as sum of all the memory operations divided by the stream bandwidth.

On MIC, even if we use all of the cores, the achieved bandwidth for SCATTER operations depends more sensitively on the input operands sizes and their distribution. Therefore, MDWIN takes the *distribution* of sizes of blocks into account. Similar to the case of GEMM, we create a lookup table for the bandwidth achieved when scattering blocks of different sizes. This lookup table comes from running a microbenchmark. Such a table appears graphically in Figure 5.4. When we SCATTER the (i, j)-th block of size $b_x \times b_y$, we estimate the time spent as $3b_x b_y / B(b_x, b_y)$, where $B(b_x, b_y)$ is the value obtained from the lookup table. Thus, $t^{(\phi)}_{\text{SCATTER}}$ is estimated as,

$$t^{(\phi)}_{\text{SCATTER}} = \sum_{i,\, j} \frac{3b_x(i) b_y(j)}{B(b_x(i), b_y(j))}. \tag{5.6}$$

5.5.1 Performance of model-driven work partitioning MDWIN

We evaluated the efficacy of MDWIN by comparing its performance against two static work partitioning schemes, denoted STATIC_0 and STATIC_1. Both STATIC_0 and STATIC_1 assign a fixed fraction, called the *offload-fraction*, of columns of $U(k)$ to MIC, to divide

[2] For scatter operation $A(i, j) \leftarrow A(i, j) - V(i, j)$, we assume two reads and one write for each element

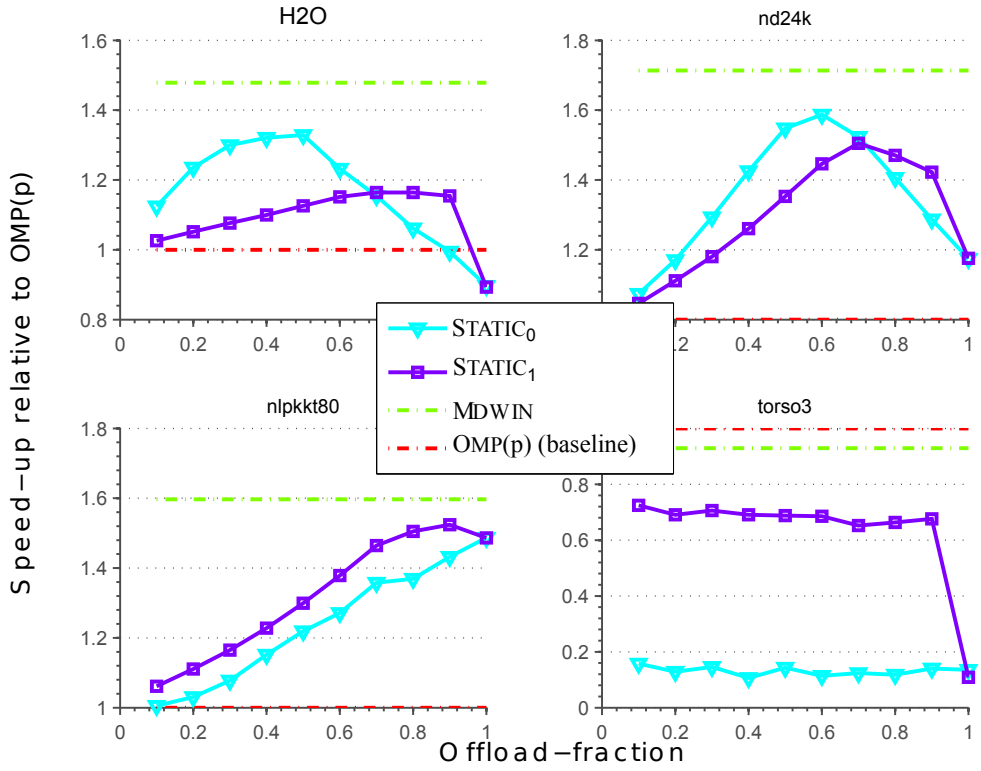


Figure 5.5: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

work between the CPU and the MIC in each iteration. In addition, STATIC_1 does not offload any work to MIC in some iteration, if operand sizes are smaller than a fixed cut-off.[3]

We used four matrices for comparison. For each matrix, we vary the offload-fraction to find its optimal value (Figure 5.5). For both STATIC_0 and STATIC_1, for different matrices, the optimal offload-fraction occurred at different values. This illustrates the main limitation of such a fixed static partitioning scheme, which is that we cannot *tune* the offload-fraction for one matrix and use it for other matrices. In addition, a bad choice of the offload-fraction may slow down the computation by 10×, e.g., in case of STATIC_0 and *torso3* combination.

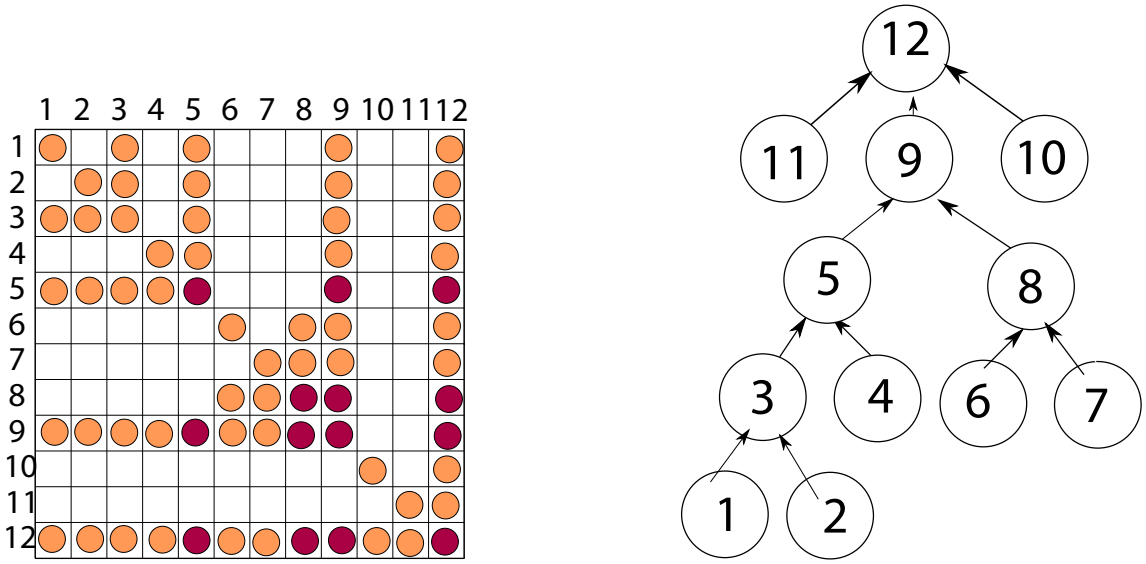
[3]We use $m_t = n_t = 512$, $k_t = 16$ as cut-offs, selected based on the relative performance of GEMM (Figure 5.3).

For all the matrices, MDWIN outperformed STATIC$_0$ and STATIC$_1$. Even in an especially difficult case (e.g., *torso3*), MDWIN incurred only a small slow down (1.1×) versus 1.4–10× of STATIC$_0$ and STATIC$_1$.

carefully implemented to reduce any overheads. Empirically, MDWIN's overhead is less than 2% of the total factorization time across our experiments (not shown here).

5.6 Limited device memory considerations

Storing large matrices may require larger than the MIC's 8 GiB of memory. Using multiple MICs is not always possible or economical. Therefore, we augment the HALO algorithm with a kind of "out-of-core" strategy that keeps a submatrix on the MIC and offloads updates of this submatrix.



(a) (b)

Figure 5.6: (a) The non-zero structure of some sparse matrix; (b) the elimination tree of the sparse matrix [5]. If only 4 panels fit on the MIC, then our heuristic keeps the panels corresponding to nodes with largest number of descendants—here, 5, 8, 9 and 12—on the MIC.

Why might such an approach work in practice? In a sparse LU factorization,

a small number of blocks are updated in many more iterations than other blocks. Therefore, the Schur-complement update of a small number of blocks can account for a large fraction of all Schur-complement update computations.

For example, consider the 12×12 sparse matrix in Figure 5.6a. The $(12, 12)$ block is updated during the Schur-complement updates of iterations 1 through 11. On the other hand, the blocks in the 1,2,4,6,7,10, and 11-th panels are not updated in any iteration's Schur-complement update. Therefore, it should be possible to offload a large fraction of the Schur-complement update operations keeping only a small fraction of matrix blocks on the MIC.

To choose these blocks, we use a heuristic based on the *elimination tree* of the sparse matrix [5]. The elimination tree, computed in any sparse LU factorization, shows which blocks will be updated in the k-th iteration's Schur-complement update. More specifically, we need only update the panels corresponding to the ancestors of the k-th node on elimination tree. For example, Figure 5.6b shows the elimination tree for the matrix of Figure 5.6a. In first iteration's Schur-complement update, only panels 3, 5, 9, and 12 need to be updated.

Conversely, the k-th panels are updated in all the iterations corresponding to descendants of k-th node in the elimination tree. Therefore, panels having the largest number of descendants are updated in largest number of iterations. Thus, our heuristic is to choose such panels. For example, in the elimination tree of Figure 5.6b, the nodes with the largest number of descendants are 5, 8, 9 and 12. Therefore, we would keep the 5, 8, 9 and 12-th panel matrices, shown in red in Figure 5.6a, on the MIC.

The best-case scenario for the above scheme is if the MIC has infinite memory, so that we would not need to consider by how much to restrict offload. Relative to this ideal scenario, Section 5.7 shows the above scheme can obtain speedups very close to the infinite-memory ideal.

5.6.1 Effect of limited MIC memory

We wish to see how well HALO performs on an accelerator whose memory is much smaller than the host's memory. To do so, we vary the fraction of the matrix in the MIC, to simulate acceleration with a small memory constraint, and see its effect on the number of flops offloaded. For this experiment, we use two matrices: (1) *nd24k* that can completely fit in the MIC's memory, and (2) *nlpkkt80* cannot.

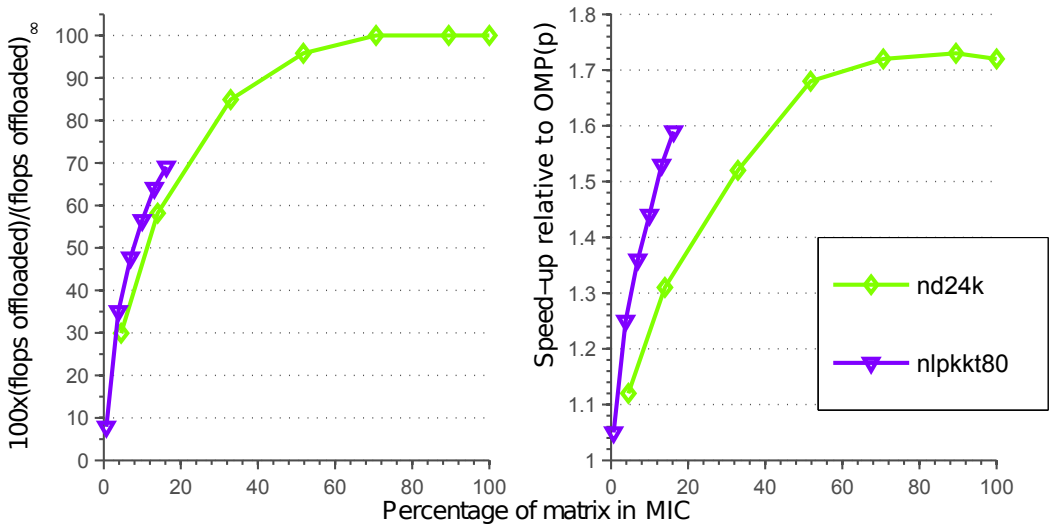


Figure 5.7: (Left) Flops offloaded to MIC as we vary the matrix kept in the device, shown as a percentage of (flops offloaded)$_\infty$. flops offloaded to MIC if it has *infinite* memory). It increases steeply with fraction of matrix kept in the MIC. (Right) The fraction of flops offloaded is directly correlated to the speed-up obtained relative to OMP(p).

In Figure 5.7 (left), the fraction of offloaded flops increases steeply as the matrix fraction stored on the MIC increases. For both the matrices, by only keeping 17% of the matrix in the MIC, we are able to offload more than 70% of the number of flops that we offload if the MIC has infinite memory. Thus, HALO can gracefully handle the relatively low memory capacity of the MIC.

In the Figure 5.7, we also show the speedup obtained. For large matrices, such as *nlpkkt80*, MIC-acceleration becomes more critical. Thanks to HALO, speedups for such a large matrix is close (within 10%) to the best small matrices case.

Table 5.1: List of Matrices used for performance evaluation.

Matrix	n	$\frac{nnz(A)}{n}$	Fill-in ratio	# Flops in factorization
atmosmodd	1,270,432	6.93	244.00	1.12E+13
audikw_1	943,695	82.28	35.01	1.13E+13
dielFilterV3real	1,102,824	80.97	14.57	1.94E+12
Ga19As19H42	133,123	66.74	180.20	1.59E+13
Geo_1438	1,437,960	41.89	85.71	3.28E+13
H2O	67,024	33.07	210.98	2.28E+12
nd24k	72,000	398.82	23.08	3.98E+12
nlpkkt80	1,062,400	26.53	141.63	3.03E+13
RM07R	381,689	98.15	74.09	2.71E+13
torso3	259,156	17.09	63.80	3.11E+11

5.7 Results on MIC accelerated systems

5.7.1 Experimental setup

Test Matrices: The matrices used in our tests are listed in Table 5.1. These matrices, taken from the University of Florida Sparse Matrix Collection, come from various real applications [21]. These matrices vary in sparsity structure, which in turn affects the sparsity of the L and U factors, the factorization time, and the overall flop rates for the factorization.

Testbeds: We used two systems in the performance evaluation: IVB20C, which is a single-node 2×10-core Ivy Bridge-EP machine with a Intel Xeon Phi co-processors; and BABBAGE, which is a 45-node 2×8-core Sandy Bridge-EP with two Xeon-Phi cards machine, located at NERSC. The key machine parameters for the two systems are listed in Table 5.2.

We used the Intel C Compiler (ICC 15.0.0) with Intel MPI Runtime Library (IMPI 5.0) and Intel Math Kernel Library (MKL) version 11.1.

In all the experiments, we used the default settings for SUPERLU_DIST: ordering

Table 5.2: Testbeds used for performance evaluation.

	Test-bed	IVB20C	BABBAGE
Host	CPU Micro-architecture	Ivy Bridge-EP	Sandy Bridge-EP
	Sockets/Cores/Threads	2/20/40	2/16/32
	Clock rate	2.80GHz	2.60GHz
	DRAM capacity	128 GB	128 GB
	Stream bandwidth	95 GB/s	72 GB/s
	Peak DP floating point performance	448 GF/s	332 GF/s
	PCIe type-Bandwidth	PCIe 2.0-8 GB/s	PCIe 2.0-8 GB/s
MIC	#MIC per node	1	2
	Clock rate	1.09 GHz	1.05 GHz
	Cores/Threads	61/244	60/240
	Stream bandwidth	163 GB/s	150 GB/s
	Peak DP floating point performance	1063 GF/s	2×1008 GF/s

via *Metis* on $|A|+|A|^T$ and static pivoting and equilibration via *MC64*. The maximum size for any supernode was set to 192. Typically, a small supernode size eases load balance among different MPI processes; therefore, we chose a small supernode size where both the GEMM and SCATTER kernels obtain reasonable performance on both CPU and MIC.

For large matrices, we limited the user allocated memory on MIC to 7 GB. For the distributed experiments, for a given number of MPI processes, we tried different combinations of $P_r \times P_c$ on unaccelerated SUPERLU_DIST, and used the best configuration when running either with and without MIC acceleration.

We studied various single node characteristics of HALO on IVB20C. For comparison, we used multithreaded SUPERLU_DIST as our baseline. This baseline uses OpenMP threads across 20 Ivy Bridge cores, and is denoted by OMP(p).[4]

[4]It is the strongest of all possible baselines to which the MIC-accelerated version can be directly compared.

5.7.2 Single node performance on IVB20C

In our single node experiment, we seek to understand gains of MIC acceleration for different matrices. We compare the following two configurations of SUPERLU_DIST on IVB20C:

- OMP(p) (Baseline)
- OMP(p)+MIC: OMP(p) added with MIC acceleration.

For these two configurations, Table 5.3 breaks down the factorization time and the obtained speedup. Overall, offloading the Schur-complement update to the MIC makes it faster by η^{sch}=0.9−1.8×, which results in an overall speed-up (η^{net}) of 0.9-1.7×. In addition to η^{sch}, overall speedup (η^{net}) also depends on the time spent in panel-factorization computations (t_{pf}), also shown in Table 5.3. In 8 out of 10 test cases, t_{pf}—being less than 20% of the baseline—is not the bottleneck.

Overall, the gains from HALO vary for different matrices. In our analysis, we treat total the factorization time (t_{omp}), the panel-factorization time (t_{pf}), and η^{sch} as independent quantities. However, they are related to the sparsity pattern of the input matrix, and can be inter-related. For example, the matrices *torso3* and *dielFilterV3real* are among the smallest in factorization time; they both also show a large t_{pf} and a small η^{sch}. Therefore, the study of the effects of sparsity-pattern on the performance of HALO is warranted.

5.7.3 Offload efficiency

We estimate the performance of HALO in the absence of any load imbalance. Load imbalance can be due to limited MIC memory, exposed PCIe communication and latency costs, or the limitations of MDWIN. Furthermore, it can cause the CPU and the MIC to idle, which would manifest as nonzero t_{cpu_idle} and t_{mic_idle} values in Table 5.3. In a hypothetical case where there are no load imbalances and PCIe

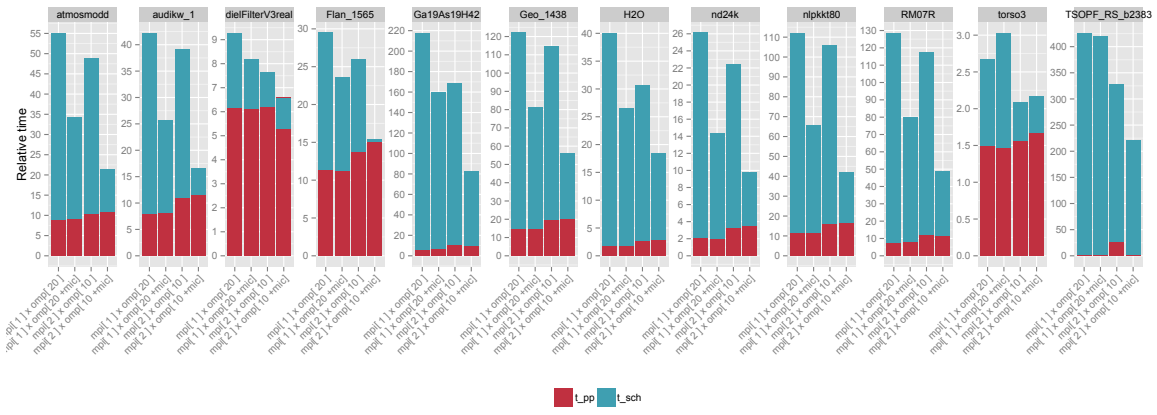


Figure 5.8: Single Node Performance on IVB20C

communication has *zero* cost (due to either hardware or software improvements), if time for factorization is t_{ideal}, then offload efficiency ξ is given by $\frac{t_{ideal}}{t_{mic}}$. To calculate offload efficiency, we estimate t_{ideal} by making a simplifying assumption, which is that the incurred load imbalance can be shared *equally* between the CPU and the MIC. In other words, in the absence of any load imbalance, the factorization time of the HALO can be reduced further by $(t_{cpu\_idle}+t_{mic\_idle})/2$. Thus,

$$\xi = 1 - \frac{t_{mic\_idle} + t_{cpu\_idle}}{2t_{mic}}. \tag{5.7}$$

This value ranges from between 0.5 (only one resource, CPU or MIC, is working and the other is completely idle) and 1.0 (both CPU and MIC are working and perfectly load-balanced). We evaluate Equation (5.7) and show it in the last column of Table 5.3. For many matrices, our implementation already achieves within 30% of the upper bound, and achieves close to 83% for *nd24k*. For bigger matrices, due to limited MIC memory, the offload efficiency hovers around 70%.

5.7.4 Single node performance on BABBAGE

Each node on BABBAGE has 2×8-core Sandy-Bridge sockets and two MICs. To use the both MICs, we spawn two MPI processes on each node and assign each MPI process a

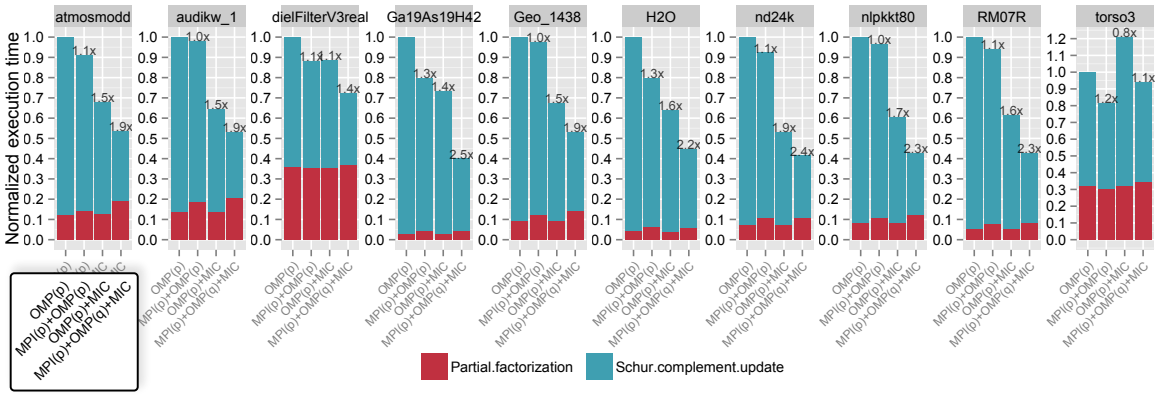


Figure 5.9: Performance for different matrices and different configuration of SUPERLU_DIST on the single node of the BABBAGE cluster. The configurations OMP(p) and MPI(p)+OMP(q) use only CPU cores, while OMP(p)+MIC and MPI(p)+OMP(q)+MIC, in addition to CPU cores, use one and two MICs, respectively. On top of each matrix×configuration bar, we show the speedup with respect to OMP(p). Overall, we obtain an additional 1.1-1.8× speedup when we use an additional MIC.

MIC. In addition to the shared memory configurations OMP(p)and OMP(p)+MIC, we also compare the following distributed memory configurations on BABBAGE:

- MPI(p)+OMP(q): One MPI process on each socket and uses OpenMP multi-threading at socket level; and
- MPI(p)+OMP(q)+MIC: MPI(p)+OMP(q)added MIC acceleration for each MPI process.

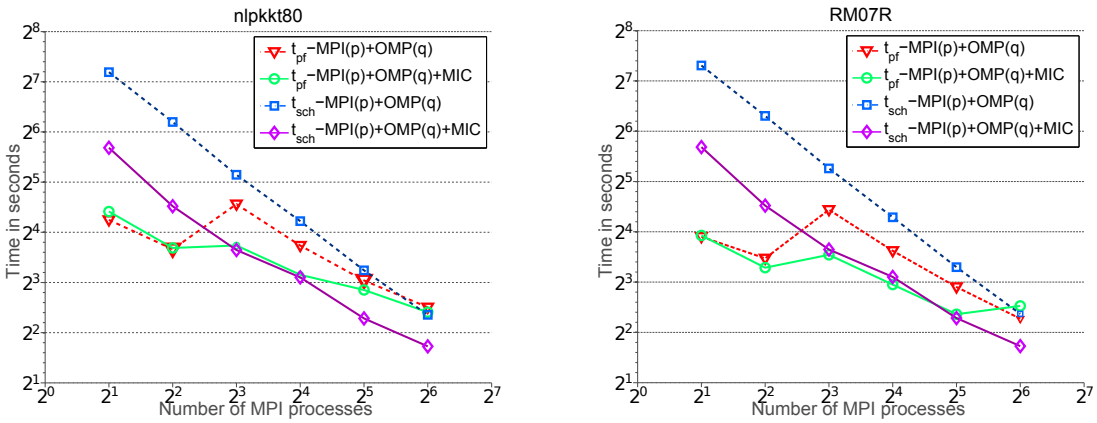
The factorization time on BABBAGE for the different configurations and matrices appears in Figure 5.9. The time for each configuration is split into the panel-factorization and the Schur-complement update phases. Note that the panel-factorization phase in MPI(p)+OMP(q) and MPI(p)+OMP(q)+MIC now include the time for MPI_Send, MPI_Recv and MPI_Wait. Therefore, t_{pf} increases when we go from shared memory to distributed memory configuration. On the other hand, the Schur-complement update phase shows better scalability in MPI(p)+OMP(q) configuration due to absence of any NUMA overheads.

Overall, when we include another MIC, we can improve the performance by additional 1.1-1.8×. This performance improvement comes from two sources. First, we

offload more computation to the MIC since relative performance of the MIC with respect to the host CPU has increased. Secondly, for large matrices such as RM07R and Ga19As9H42, where only a small fraction of the matrix can fit in one MIC, the increased fraction of the matrix fits in two MICs. Thus, more computation can be offloaded to the MIC.

5.7.5 Strong scaling on BABBAGE

We study strong scaling of HALO to understand the effects of MIC-acceleration when we scale SUPERLU_DIST to a large number of nodes. For this experiment, we use two matrices, *RM07R* and *nlpkkt80*, and we scale up to 32 nodes in two MPI processes per node configuration.



(a) (b)

Figure 5.10: Strong scaling of the baseline (MPI(p)+OMP(q)) and MIC-accelerated (MPI(p)+OMP(q)+MIC) configurations on BABBAGE. For both matrices, the panel factorization phase (t_{pf}) does not scale as well as the Schur-complement update. Therefore, at a large number of processes, panel factorization will become bottleneck.

Figure 5.10 shows the scaling of the panel-factorization and Schur-complement update phases of computation. As a reference, we also give the scaling results for the baseline SUPERLU_DIST.

The Schur-complement update phase for both matrices and both configurations scales almost linearly with number of MPI processes, while the panel-factorization

phase does not. Thus, at 64 MPI processes, the cost of panel-factorization dominates. HALO's scalability is affected even more than the baseline, since its Schur-complement update cost is smaller.

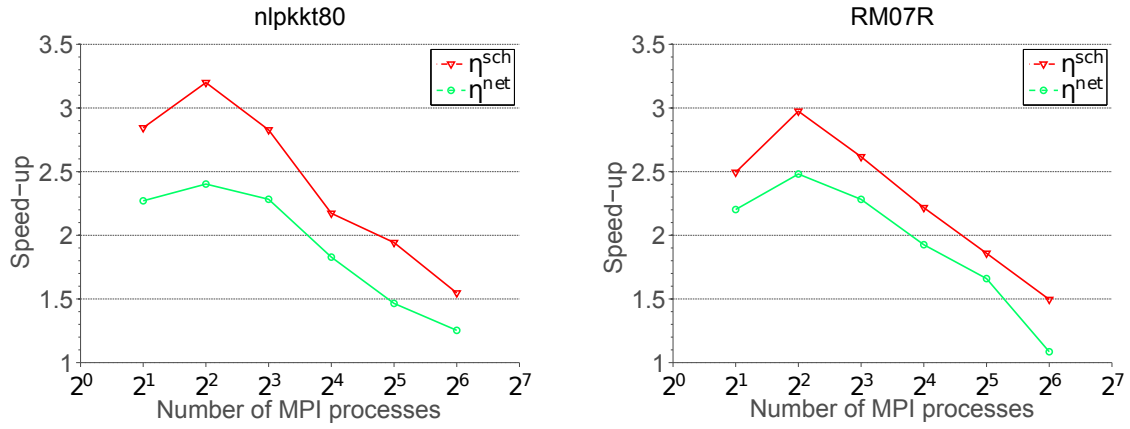


Figure 5.11: Strong scaling on BABBAGE: speedup of Schur-complement update η^{sch} and overall speedup η^{sch} of MPI(p)+OMP(q)+MIC with respect to MPI(p)+OMP(q) for different number of MPI processes.

Figure 5.11 shows the speedup of HALO over the baseline. The speedup in the Schur-complement update phase, η^{sch}, increases when we go from two to four MPI processes. This increased speedup is due to an increased fraction of the matrix residing on the MICs. When we go beyond four processes, the per iteration work of the Schur-complement update phase decreases; thus, η^{sch} also gracefully decreases, reducing to about 1.5× at 64 MPI processes. This η^{sch}, however, results in an overall speedup (η^{net}) of only 1 to 1.25×, as the panel-factorization phase dominates the total time at 64 MPI processes.

5.8 Results on GPU accelerated systems

We also implemented the HALO algorithm for GPU based heterogeneous systems. The algorithm-wise, GPU implementation reflected all the optimization that we discussed in the context of the MIC based accelerator. The difference between GPU and

MIC arises for SCATTER implementations. In principle, we could use the SCATTER code for host multicore for MIC without modification, though not for GPUs. However, for getting good performance, MIC and GPU required a similar amount of effort. We evaluated the performance of GPU-HALO on two testbeds shown in Table 5.4.

5.8.1 Single Node Performance

The single node performance of GPU-HALO on CONDESAand TITANare shown in Figure 5.12 and Figure 5.13, respectively. Qualitatively, we obtain performance on GPU based is similar to what we obtain on MIC based platforms for different matrices. However, the host multicore is relatively lower floating-point operations per second for both CONDESAand TITAN, therefore the "speed-up" of GPU accelerated SUPERLU_DIST with respect to the host multicore appears relatively higher.

5.8.2 Strong scaling on TITAN

We show the strong scaling results for two matrices nlpkkt80 and RM07R on Figure 5.14 and Figure 5.15 respectively. Again, the results on TITANare qualitatively similar to the MIC-accelerated BABBAGEsystem. Up to 16 nodes of TITAN, we obtain a sustained speed up of $\geq 1.7\times$ with respect to multicore CPU. As we go beyond 16 nodes, the benefit of GPU acceleration begins to diminish. Primarily at the larger node counts, panel- factorization become as costly as the Schur-complement update. Since HALO accelerates the Schur-complement update computation, thus the cross-over point when panel-factorization becomes as costly as Schur-complement update, arrives at smaller node count than the multicore baseline. Thus, HALO improves the node utilization at smaller node counts, but it may not improve the performance when panel-factorization has become the bottleneck. This observation is consistent with our results on MIC accelerated cluster BABBAGE.

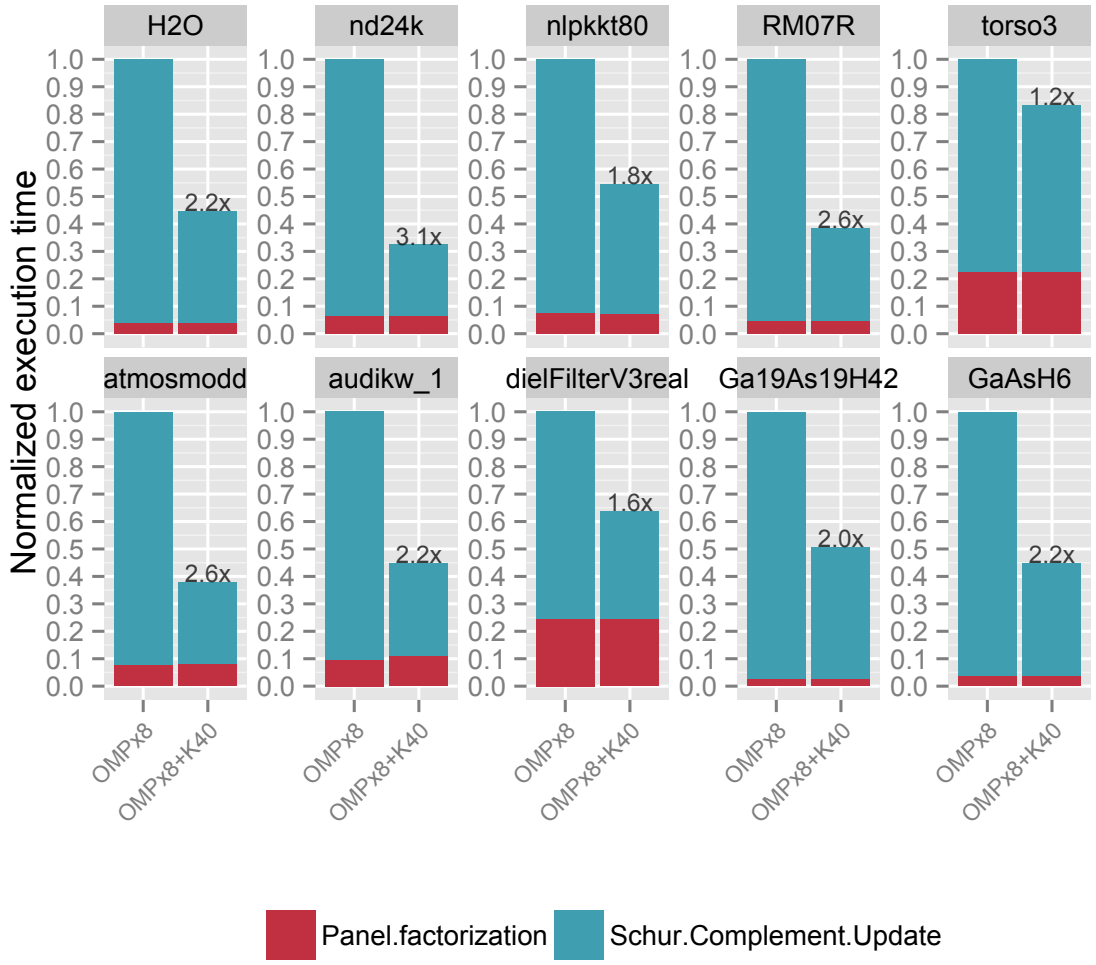


Figure 5.12: Single Node Performance of GPU-HALO on Condesa

5.9 Conclusion and new challenges

None of the important technical components of HALO are specific to MIC, meaning the same ideas should extend naturally to GPU-based clusters or other heterogeneous node architectures. However, architectural differences between GPUs and MIC may result in different relative performance profiles for different operand sizes. A combined software infrastructure that can exploit either or both types of accelerators is a natural extension.

At a sufficiently large number of MPI processes, one should expect the time spent

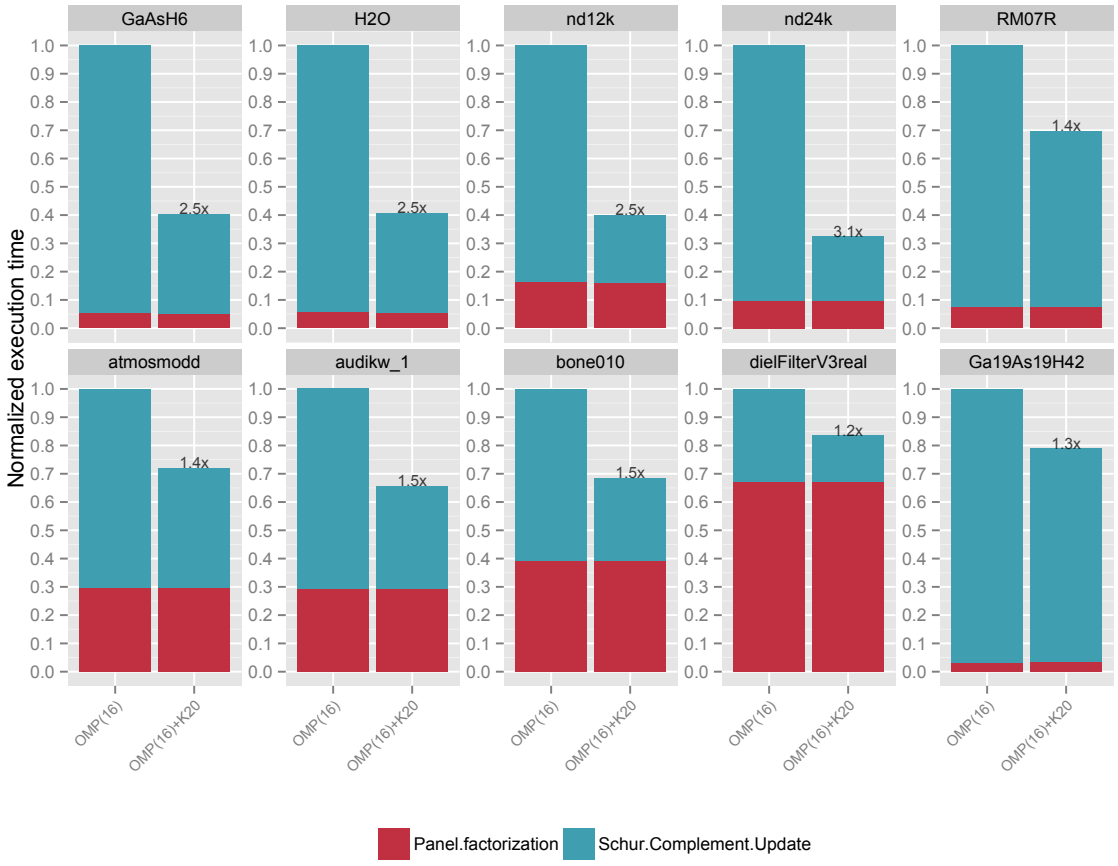


Figure 5.13: Performance for different matrices and different configuration of SUPERLU_DIST with gpu-HALO

in panel-factorization phase to begin to dominate. This would happen primarily because of increasing load imbalance among the processes, which leads to increases in MPI_Wait and MPI_Recv times. This problem could be avoided if the bottleneck process could assign more work to its accelerator, as a way of reducing the apparent load imbalance. However, knowing precisely when to do so would require a scheme to estimate time at a *global* level, which would have to include modeling of potential load imbalance and MPI communication costs. This topic is another excellent one for future work.

Part of the proposed scheme targets the relatively smaller memory capacities of accelerators compared to their hosts. However, an intrinsic problem in sparse

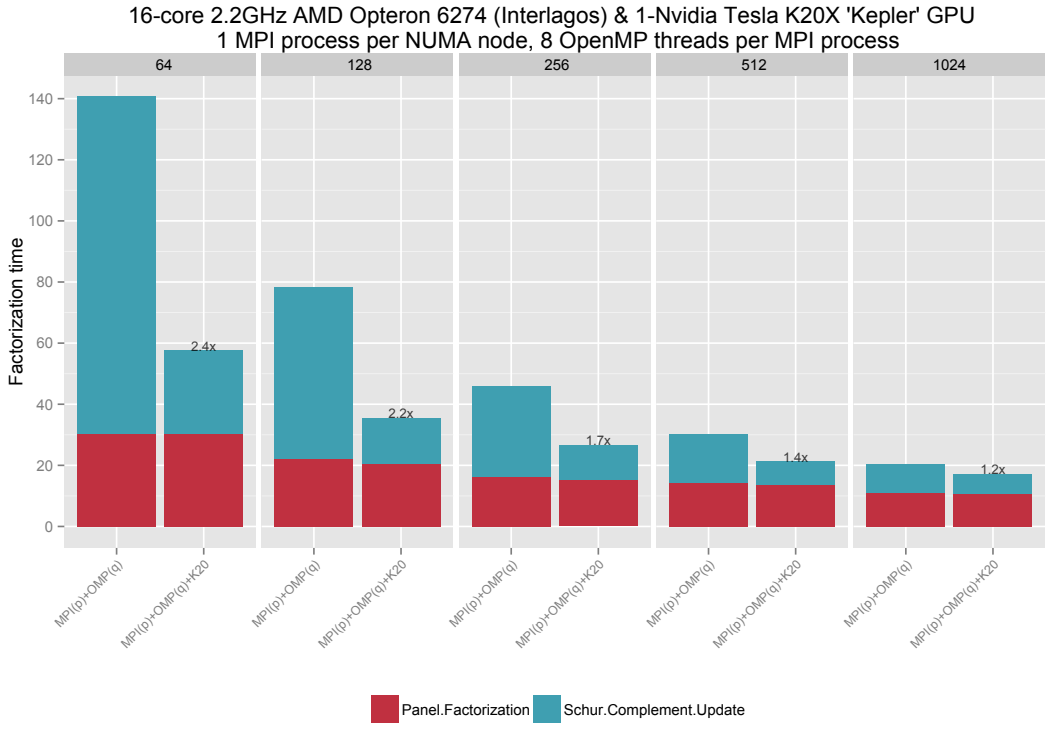


Figure 5.14: Strong scaling on Titan Cluster for nlpkkt80

LU is that the per-process memory requirement tends to increase as the number of processes increases. In this sense, accelerators can help by decreasing the need for more MPI processes. Nevertheless, a more formal and precise understanding of this issue would be needed to scale sparse direct solvers to the next level.

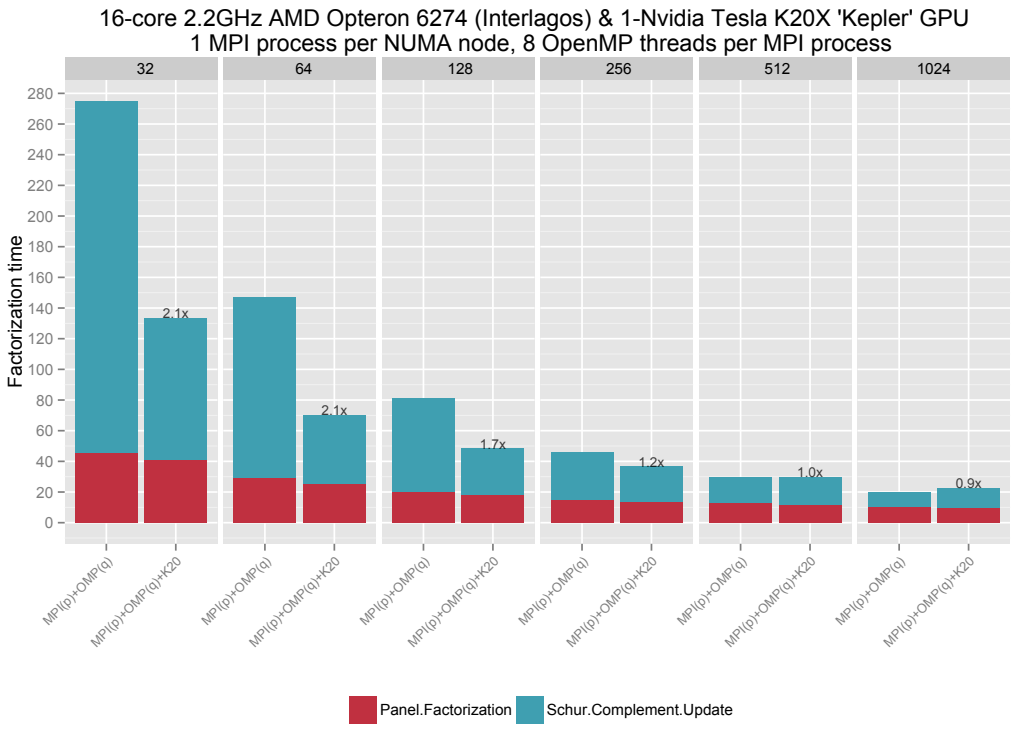


Figure 5.15: Strong scaling on Titan Cluster for RM07R

		Factorization time (in sec.)			Speed-up		Miscellaneous time (OMP(p)+MIC)			
	Matrix	OMP(p) (t_{omp})	OMP(p)+MIC (t_{mic})	Panel-fact. (t_{pf})‡	Schur-Comp. Update (η^{sch})	Overall (η^{net})	t_{cpu_idle}†	t_{mic_idle}†	t_{pcie}†	Offload efficiency(ξ)
Fits in MIC memory	H2O	41.9	28.3	4.3%	1.5	1.5	6.12%	32.4%	9.7%	80.7%
	nd24k	28.2	16.4	7.3%	1.8	1.7	4.9%	29.4%	7.6%	82.85%
	torso3	4.2	4.5	35.2%	0.9	0.9	7.9%	72.6%	4.8 %	59.7%
Does not fit in MIC memory	atmosmodd	64.2	43.4	14.1%	1.6	1.5	7.35%	50.8%	5.7%	70.3%
	audikw_1	50.3	33.7	16.1%	1.6	1.5	6.37%	49.5%	5.7%	72.4%
	dielFilterV3real	15.5	14.3	39.5%	1.1	1.1	2.7%	74.8%	6.4%	62.3%
	Ga19As19H42	224.3	165.8	2.9%	1.4	1.4	1.8%	59.6%	2.1%	69.3%
	Geo_1438	136.6	96.1	10.8%	1.5	1.4	1.34%	67.6%	2.7%	65.4%
	nlpkkt80	123.9	77.6	9.5%	1.7	1.6	0.44%	64.0%	2.9%	67.8%
	RM07R	136.3	87.6	5.7%	1.6	1.6	5.0%	54.9%	6.1%	70.0%

Table 5.3: Single node performance of MIC accelerated SUPERLU_DIST

Factorization time of OMP(p) and OMP(p)+MIC for different matrices on the single node IVB20Csystem. ‡: Time shown is a percentage of t_{omp}. †: Time shown is a percentage of the t_{mic}. Overall speedup η^{net} ranges from 0.9 to 1.7×, and depends on the unaccelerated fraction (t_{pf}) and speedup obtained in MIC-accelerated Schur-complement update (η^{sch}). The last four columns show the idle time of the CPU and the MIC, the PCIe transfer time, and the estimated offload efficiency.

Table 5.4: GPU-Testbeds used for performance evaluation.

	Test-bed	CONDESA	TITAN
Host	CPU Micro-architecture	Sandy Bridge-EP	AMD “interlagos”
	Sockets/Cores/Threads	1/8/8	1/8/16
	Clock rate	1.80GHz	2.60GHz
	DRAM capacity	128 GB	32 GB
	Stream bandwidth	51 GB/s	72 GB/s
	Peak DP floating point performance	128 GF/s	216 GF/s
	PCIe type-Bandwidth	PCIe 2.0-8 GB/s	PCIe 2.0-8 GB/s
GPU	# GPU per node	1	1
	Clock rate	745MHz	706MHz
	# stream processors	2880	2496
	Stream bandwidth	288 Gbytes/sec	208 Gbytes/sec
	Peak DP floating point performance	1.43 TFLOPS	1.17 TFLOPS
	Model Name	Tesla K40 “Kepler”	Tesla K20“Kepler”
	Memory Capacity	12GB	5GB

CHAPTER 6

A COMMUNICATION-AVOIDING DISTRIBUTED SPARSE LU

6.1 Problem Statement

This chapter moves beyond on-node scaling and revisits the distributed memory algorithm underlying SUPERLU_DIST. We look for opportunities to further reduce network communication, especially in the strong scaling regime where communication must eventually become relatively more expensive.

The replication technique of the previous chapter is, in fact, an instance of applying *communication-avoiding* techniques. In that body of research, one critical strategy is to shrink the amount of data transferred through redundant computation, data replication, or both. There are several examples for dense linear algebra [32, 33, 34], including some for dense LU [30, 35]. However, precisely how to apply communication-avoiding methods to sparse LU had thus far been open.

This chapter describes our design and implementation of the first such method, which we refer to as a *3D sparse LU factorization algorithm*. It is so-named for two reasons, both inspired by the 2.5D *dense* LU algorithm [30]. First, it uses a 3D logical process grid, instead of the 2D process grid that is the state-of-the-art in sparse LU. Secondly, and replicates data to reduce both the number of messages and the volume of communication.

In addition, the parallelism of all sparse LU methods is captured by an *elimination tree*, which our method uses to efficiently map the problem to the 3D process grid. As a result, our algorithm not only reduces communication but also shrinks the critical path of the factorization—a feature that does not apply to the 2.5D dense LU case. For matrices with planar graph structure, such as planar grids and meshes,

our 3D sparse LU algorithm's critical path is $\mathcal{O}(n/\log n)$ whereas a state-of-the-art 2D algorithm's is $\mathcal{O}(n)$.

Briefly, here is how 3D sparse LU works. First, consider the 3D process grid as a collection of 2D grids. We divide the elimination tree into independent subtrees and a common ancestor tree of all the subtrees. Factoring each subtree is independent, but each factorization updates the common ancestor tree. We map the factorization of each subtree to a 2D grid and replicate the common ancestor on all process grids. Each 2D grid factorizes its subtree and uses its copy of the common ancestors to perform Schur-complement updates. We then reduce these copies onto a single grid, where it is factored in a 2D fashion.

We implement this scheme on top of SUPERLU_DIST, using a hybrid MPI+OpenMP programming model. We measure performance on a wide range of matrices in both 2D and 3D process grid configurations. (The baseline is 2D SUPERLU_DIST.) In the best case, we observe speedups of 27× over the best 2D process-grid configuration at a cost of 1.7× the memory. We observe that our new algorithm can use up to 16× as many processors for the same problem size with continued time reduction, thereby confirming its potential to enhance strong scaling significantly. We also derive performance models to help understand how the speed of the new algorithm depends on the sparsity structure of the matrix and process grid configuration.

6.2 Limitations of the 2D algorithm

The 2D algorithm scales well up to a certain number of processes, beyond which the cost of data transfer starts to dominate the cost of computation. Moreover, at a large process counts, the effect of load imbalance becomes more prominent. So after a certain number of processes, we see that adding more processes can cause a slowdown in the factorization time. Fundamentally, the 2D algorithm suffers from the following two major limitations:

1. *Sequential Schur-complement update*. For a given block, only one process can perform the Schur-complement update in the 2D algorithm. So despite abundant tree-level parallelism, the 2D algorithm must perform all Schur-complement updates sequentially.

2. *Fixed latency cost*. Almost all processes participate in the factorization of all the supernodes. So the latency of various communication kernels does not decrease with increasing number of processors.

How can we perform the updates on a given block A_{ij} in parallel by two different processes? The 2D algorithm uses an owner-only update policy. So, the Schur-complement update on a given block is sequential. This fact motivates our approach of replicating some blocks of A on different processes. Doing so allows the Schur-complement updates on those blocks to proceed in parallel. But how do we choose such blocks and processes to replicate?

6.3 A 3D LU algorithm for sparse matrices

6.3.1 The 3×3 block sparse case

We can use the etree to decide how to replicate data. Consider the 3×3 block sparse matrix shown in Figure 3.3a and its etree. After factoring blocks 1 and 2, the block A_{33} needs to update from both according to

$$A_{33} = A^0{}_{33} - L_{31}U_{13} - L_{32}A_{23}.$$

Suppose we replicate, keeping two copies of the block A_{33}. The first copy accumulates $A^0{}_{33} - L_{31}U_{13}$ from the factorization of the block 1; the second copy accumulates $-L_{32}U_{23}$ from block 2. We then sum the two copies to get final form of A_{33} before factoring it. Thus, the replication of A_{33} allows parallel Schur-complement update of

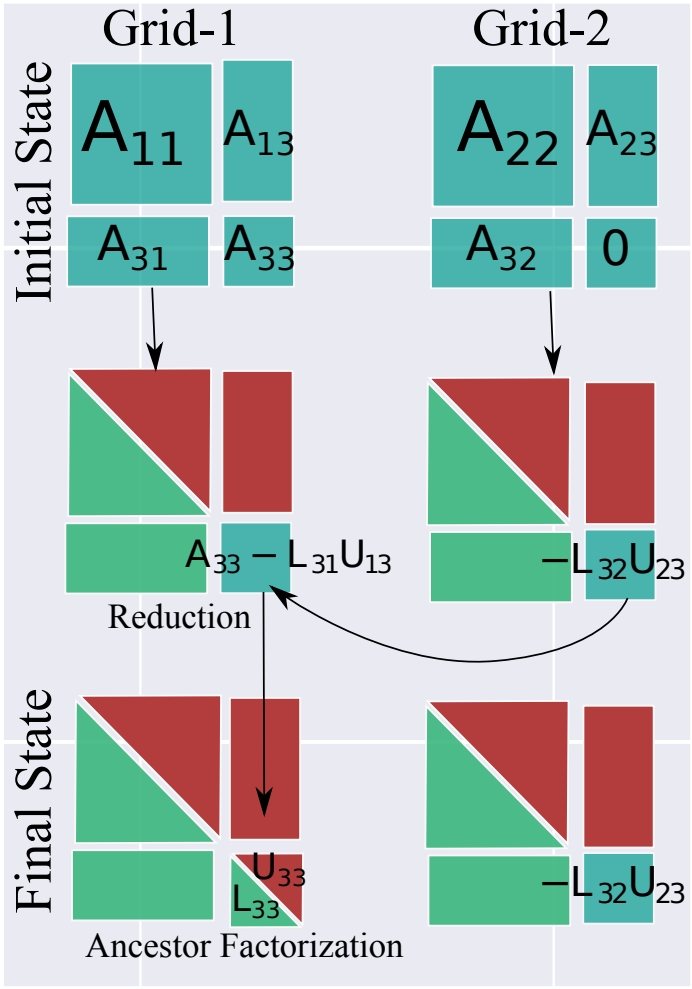


Figure 6.1: How 3D sparse LU works for a block sparse matrix A (Figure 3.3a) using 2 process-grids. The sparse blocks 1 and 2, and their panels, reside on grid 1 and grid 2, respectively. Block 3 is replicated in both the grids and is initialized with A_{33} and 0 on grid 1 and grid 2, respectively (the initial state). The two grids factor their respective blocks and Schur-update their copies of the block 3. Then, we reduce the 3rd block from both grids onto grid-1, which is then factored on grid 1. Lastly, the L and U factors are distributed among the two process grids (final state).

block A_{33}. Figure 6.1 shows the timeline of this process.

Formally, we carry out this process as follows. Let E be the etree of the matrix A. We partition E into two independent subtrees, C_1 and C_2, and a common parent S (Figure 6.2a). We partition A into $A^1 = A(C_1) \bigcup A(S)$ and $A^2 = A(C_2) \bigcup A(S)$ (Figures 6.2c and 6.2e). We factor A^1 and A^2 in two 2D process grids, grid-1 and grid-2. In grid-2, we initialize the blocks of $A(S)$ with zeros. Grid-1 and grid-2 factor C_1 and C_2 in parallel and update their copy of $A(S)$. After the factorization, they synchronize, and grid-2 sends its copy of $A(S)$ to grid-1:

$$A^1(S) = A^1(S) + A^2(S)$$

Then the grid-1 factors the updated copy of $A(S)$.

The two process grids only need to communicate once. Below, we show that this is a small fraction of the total communication. Furthermore, now each process factors a smaller number of supernodes, which reduces latency.

6.3.2 General Case

Suppose we want to use four 2D grids instead of two. We can divide the etree in one more level. For instance, in Figure 6.3, we have a two-level etree that we divide into four partial etrees. The root (node-6) of the etree is replicated in all the grids. In the first level, we replicate node 2 on grids 0 and 1, and node 5 on grids 3 and 4. In the second level, all the nodes lie on only one grid.

Process grid 0 and 1 synchronize after they have factored nodes 0 and 1, respectively. Then process grids 0 and 1, reduce all the common ancestor nodes, namely node 2 and 6 on grid 0. Similarly, process grids 2 and 3 synchronize after they have factored nodes 3 and 4 respectively. Then process grids 2 and 3 reduce all the common ancestor nodes, namely nodes 5 and 6 on grid 2.

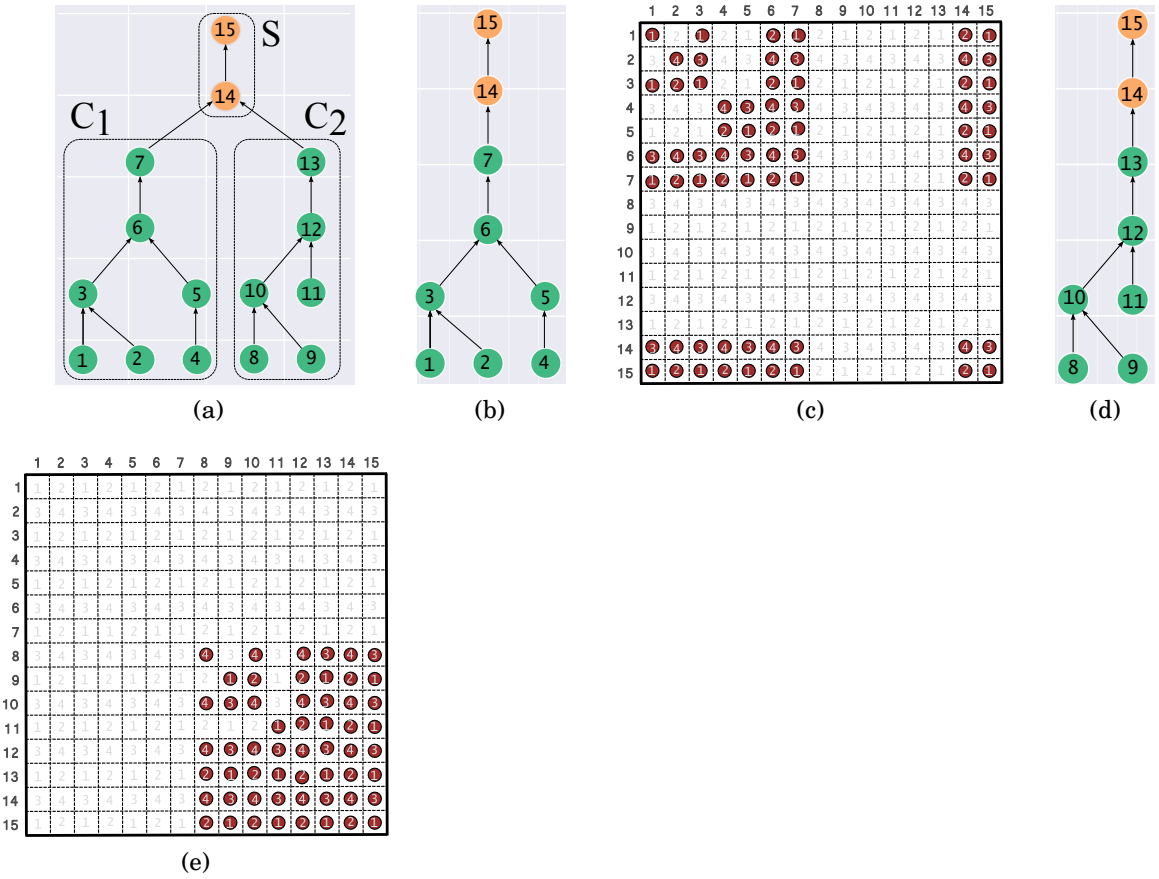


Figure 6.2: Data distribution in 3D sparse LU algorithm. The elimination tree of the block sparse matrix in Figure 3.7a is divided into common ancestor C and subtrees S_1 and S_2 as shown in Figure 6.2a. The S_1 and S_2 subtree reside and factored in process grid-1 and grid-2 respectively. Whereas C is replicated in both the $2D$ grids. The Figure 6.2b and Figure 6.2c show the local elimination tree of the grid-1 and the data distributed in grid-1,respectively; and Figure 6.2d and Figure 6.2e show the same for grid-2.

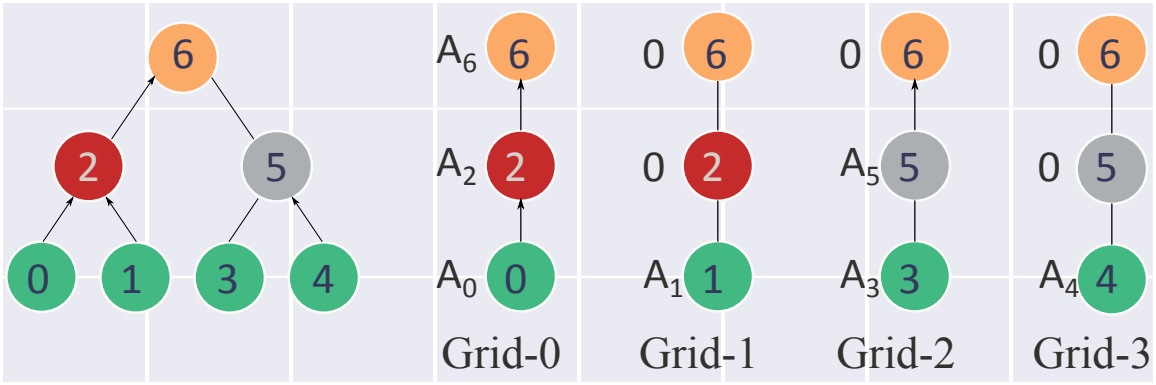


Figure 6.3:
A two-level partition of the elimination tree and its mapping into 4 process grids. Here A_i represents the diagonal block matrix along with its panels $A_i = \{A_{ii}, A_{i,:}, A_{:,i}\}$

Algorithm 6 3D Sparse LU Algorithm

function dSparseLU3D(A,E_f):
  for lvl in $l:0$ **do**:
    if $p_z = k2^{l-\text{lvl}}$ for some integer k **then**:
      dSparseLU2D(A,E_f[lvl])
      if lvl > 0 **then**:
        if $k \mod 2 \equiv 0$ **then**: ▷ Note $p_z = k2^{l-\text{lvl}}$
          dest $= p_z$
          src $= p_z + 2^{l-\text{lvl}}$
        else:
          src $= p_z$
          dest $= p_z - 2^{l-\text{lvl}}$
        for l_a in lvl $-1:0$ **do**:
          for $s \in E_f[l_a]$ **do**:
            if $p_z =$ src **then**:
              Send A_s^{src} to dest
            else:
              Receive A_s^{src} from src
              $A_s^{\text{dest}} = A_s^{\text{dest}} + A_s^{\text{src}}$
end function

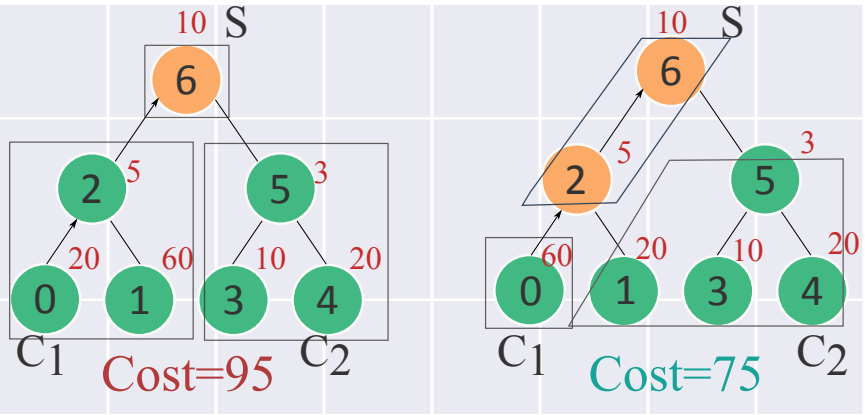


Figure 6.4: Inter-grid load balancing: an unbalanced elimination tree with 2 ways of mapping nested dissection and a greedy heuristic, and the cost of factorization in the critical path, whose length is $T(E) = T(S) + \max\{T(C_1),\ T(C_2)\}$. The cost of factorization of each node is shown in red.

In the second step, only grid 0 and grid 2 are active. They factor nodes 2 and 5, and they reduce the updates on node 6 to grid 0. And in the last step, grid 0 factors node 6. We can generalize this process for any $P_z = 2^l$, which is Algorithm 6.

6.3.3 Inter-grid Load Balancing

When the sub-trees at the top level are unbalanced, we may further divide the sub-trees to another level to get better balance. For instance, in Figure 6.4, we show an elimination tree with the cost of factorization of each node. This tree is unbalanced at the top level. The ND ordering (shown Figure 6.4-left) to partition the etree is sub-optimal. We show a better partition, which divides the subtrees by another level, in Figure 6.4-right. This partition has a smaller critical path cost of 75 units versus the ND partition that gives a critical path of length 95 units. In some cases, we may need to divide one of the subtrees even further to obtain the desired balance. We use a greedy heuristic to find a partition so that $T(S) + \max\{T(C_1),\ T(C_2)\}$ is minimized, where $T(C)$ is the cost of factoring nodes in the subtree C. However, we do not know the cost of factoring each node. We use the number of floating-point operations in factoring of a node as a heuristic cost function $T(C)$.

Algorithm 7 Load Balancing Forest Partition

function PARTITIONFOREST(E_f,ϵ):
  $E_f = \{R_1, R_2, \ldots R_k\}$, where $R_1, R_2, \ldots R_k$ are roots of the tree in the forest E_f
  $S \leftarrow \emptyset$
  $\{C_1, C_2\} \leftarrow$ partition2(E_f) ▷ divide E_f into two sets with almost equal weight
  $T^*(E_f) \leftarrow T(S) + (T(C_1) + T(C_2))/2$
  $T^0(E_f) \leftarrow T(S) + \max(T(C_1), T(C_2))$
  $i \leftarrow 0$
  $P_i \leftarrow \{S, C_1, C_2\}$
  while $T^0(E_f) \geq (1+\epsilon)T^*(E_f)$ **do**
    Let $R_k \in E_f$ be such that$T(R_k) \geq T(R_i) \forall R_i \in E_f$
    if No such R_k exists **then**
      return P_i
    $\{S_r, C_{1r}, C_{2r}\} \leftarrow$ TOPPARTITION(R_k)
    $S \leftarrow S \bigcup S_r$
    $E_f \leftarrow E_f \bigcup \{C_{1r}, C_{2r}\} - S_r$
    $\{C_1, C_2\} \leftarrow$ partition2(E_f)
    $T^*(E_f) \leftarrow T(S) + (T(C_1) + T(C_2))/2$
    $i \leftarrow i + 1$
    $T^i(E_f) \leftarrow T(S) + \max(T(C_1), T(C_2))$
    if $T^{i-1}(E_F) \leq T^i(E_F)$ **then**
      return P_{i-1}
  return P_i
end function

Elimination tree-forest E_f: Our greedy heuristic gives a partition of the etree E that can have multiple disjoint subtrees as a *node*. For instance, in the right partition of Figure 6.4, the second child C_2 consists of two unconnected components. So the final partition of the etree is a tree of forests, which we call the *elimination tree-forest* E_f. The E_f obeys the same dependency rules as E. The previous discussions of etree partitioning and mapping to grids remains the same for E_f as well.

We present the pseudocode of our heuristic in Algorithm 7. The input to Algorithm 7 is elimination tree forest E_f, which is list of root node R_k of the trees in E_f. The output of the Algorithm 7 is the partition $\{S, C_1, C_2\}$, where S is the separator forest, and C_1, C_2 are children forests. In the begining S is empty. In every iteration, we try to partition E_f into two sets C_1 and C_2, so that they $|T(C_1) - T(C_2)|$ is minimized using a call to function partition2. If the we do not have desired balance between C_1 and C_2, we choose the a tree $R_k \in E_f$ with the largest $T(R_k)$. We find the top level partition $\{S_r, C_{r1}, C_{r2}\}$ of R_k. We update the ancestor set S as $S \leftarrow S \cup S_r$, and E_f as $E_f \leftarrow E_f \cup C_{r1} \cup C_{r2} - R_k$. In the next iteration, we use updated E_f to find better balanced partition. The iteration terminates when we have achieved desired balance between C_1 and C_2. When $P_z > 2$, then we call Algorithm 7 on children forests C_1 and C_2 recursively to get desired number of partitions.

The pseudocode of the 3D sparse LU factorization appears in Algorithm 6. The parameter $P_z = 2^l$ is the number of 2D process grids, i.e., P_z is the number of processes in the "z-dimension" of the 3D process grid. And E_f is the elimination tree-forest, the output of our load-balance heuristic. Each process grid only stores forests that resides the grid, and the each forest is stored as list of nodes. We use an indexing scheme like the one illustrated in Figure 6.3. The factorization progresses from leaves lvl = 0 to the root lvl = l. We use another variable ilvl = l – lvl to simplify lengthy index expressions in Algorithm 6. The two main subroutines invoked at any level are SUPERLU_DIST_2D and Ancestor-Reduction.

1. SUPERLU_DIST_2D(A, node_list): *My* process grid performs the 2*D* factorization of nodes in the node_list on my copy of the matrix A. The forest E_f[lvl] is passed on to SUPERLU_DIST_2D as a list of supernodes. Since we use SUPERLU_DIST as the baseline data structure, in our implementation SUPERLU_DIST_2D is a call to modified factorization routine of SUPERLU_DIST.

2. *Ancestor-Reduction*: After the factorization of level-i, we reduce the nodes of the ancestor matrix before factoring the next level. In the i-th level's reduction, the receiver is $k2^{l-i+1}$-th process grid and the sender is $(2k+1)2^{l-i}$-th process grid, for some integer k. The process in the 2D grid which owns a block $A_{i,j}$ has the same (x,y) coordinate in both sender and receiver grids. So communication in the ancestor-reduction step is point-to-point and takes places along the z-axis in the 3D process grid.

Aside from these two steps in Algorithm 6, the rest are index calculations.

6.4 Communication Costs of 2D and 3D LU Factorization Algorithm

How well Algorithm 6 performs relative to the baseline depends on the sparsity pattern of the matrix. However, we can derive analytical expressions of performance on certain model problems, and thereby gain some insight into the algorithm's behavior. Our analysis considers two types of input matrices. The first are associated with planar graphs, such as those that arise when discretizing partial differential equations (PDEs) on 2D domains. The second type are those that arise with 3D PDEs, which have a "well-shaped" geometry but are non-planar.

Below, we derive expressions specifically for memory use, communication volume, and message latency (number of messages) for the baseline SUPERLU_DIST algorithm when using a 2D process grid, given a general input matrix. Then, we give expressions for both the 2D and 3D algorithms, specifically for the planar (2D ge-

ometry) and non-planar (3D geometry) model problems. These are summarized in Table 6.1.

To help distinguish the 2D and 3D *algorithms*, which use 2D and 3D process grids, from the 2D and 3D *model problems*, which have 2D or 3D geometries, we will use "planar" and "non-planar" to refer to the model problems.

6.4.1 2D sparse LU with a generic sparse matrix

Consider the factorization of a sparse matrix A of dimension n and its elimination tree E. For simplicity, assume that E is balanced at each level. Also assume that the levels in E are indexed from top to bottom: the root has a level or index value of 0, and the later levels range from 1 to nlevel, where nlevel + 1 is the number of levels in E. Let the supernode size in level-i have dimension n_i. The i-th level has 2^i nodes.

Per-process memory

The LU factors of the separator nodes, which are usually dense, account for most of the storage. Thus, each node at level-i requires a memory of n_i^2. Further suppose that SUPERLU_DIST, which uses a 2D block cyclic layout, distributes the factors evenly across P processors. Then the per-process memory, M, required to store all the LU factors is

$$M \approx \frac{1}{P} \sum_{i=0}^{\text{nlevel}} 2^i n_i^2. \tag{6.1}$$

Per-process communication volume

Per-process communication volume in the factorization for a dense matrix of size n on a $2D$ process grid, without any data replication, is given by $\mathcal{O}(n^2/\sqrt{P})$ [30]. To estimate the communication involved in the sparse factorization, we only consider the factorization of separator nodes. In this case, the per-process communication of

Table 6.1: Asymptotic memory, communication, and latency costs for 2D and 3D Sparse LU algorithm

Parameter	2D PDE		
	dSparseLU2D	dSparseLU3D	dSparseLU3D $P_z = \mathcal{O}(\log n)$
Memory per process (M)	$\mathcal{O}\left(\frac{n}{P}\log n\right)$	$\mathcal{O}\left(\frac{n}{P}(\log(\frac{n}{P_z}) + P_z)\right)$	$\mathcal{O}\left(\frac{n}{P}\log n\right)$
Communication per process (W)‡	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\log n\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}(2\sqrt{P_z} + \frac{\log n}{\sqrt{P_z}}) + \frac{nP_z}{P}\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)^{\dagger}$
Latency	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z} + \sqrt{n}\right)$	$\mathcal{O}\left(\frac{n}{\log n}\right)$

† ... when $P \gg \log n$.

‡ ... on the critical path of factorization. The average per-process communication is $\mathcal{O}\left(\frac{n\log n}{P_z\sqrt{P_{XY}}}\right)$. For $P_z = \mathcal{O}(\log n)$, both are asymptotically the same and equal to $\mathcal{O}\left(\frac{n}{\sqrt{P_{XY}}}\right) = \mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)$.

sparse LU on a $2D$ process grid is

$$W \approx \sum_{i=0}^{nlevel} 2^i \frac{n_i^2}{\sqrt{P}} = \mathcal{O}\left(\sqrt{P}M\right). \tag{6.2}$$

Latency

In the $2D$ sparse LU algorithm, each process participates in the factorization of every supernode of the sparse matrix. Thus, the number of steps for factorization is $\mathcal{O}(n)$, and the latency L (number of messages) must also scale that way, i.e.,

$$L = \mathcal{O}(n). \tag{6.3}$$

6.4.2 Planar input graphs

For a planar graph with n vertices, we can find a separator of size $\mathcal{O}\left(\sqrt{n}\right)$ in $\mathcal{O}(n)$ time [36]. This result also holds for other classes of graphs, like graphs with bounded

Table 6.2: Asymptotic memory, communication, and latency costs for 2D and 3D Sparse LU algorithm

	3D PDE	
Parameter	dSparseLU2D	dSparseLU3D
Memory per process (M)	$\mathcal{O}\left(\frac{n^{\frac{4}{3}}}{P}\right)$	$\mathcal{O}\left(\frac{n^{4/3}}{P}\left(\kappa P_z + \frac{1}{P_z^{1/3}}\right)\right)$
Communication per process (W)‡	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\right)$	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\left(\kappa_1\sqrt{P_z} + \frac{1-\kappa_1}{P_z^{4/3}}\right)\right)$
Latency	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z^{2/3}} + \kappa_0 n^{2/3}\right)$

genus and graphs with excluded minors [37, 38].

The separator divides the graph into two almost equal halves with $\sim n/2$ vertices each. These subgraphs can further be divided into two almost equal halves with a separator of size $\sim \sqrt{n/2}$. So the separator in the first level is of size $\sim \sqrt{n/2}$ and subsequently, size of separator in i-th level is $\sim \sqrt{n/2^i}$. This approximation is good when $n/2^i \gg 1$. The number of levels in the elimination tree is $\sim \log n$.

Per-process memory

We calculate the memory per-process for 2D sparse LU using Equation (6.1). For a planar graph, $n_i = \sqrt{n/2^i}$ and there are $\log n$ levels, so the per-process memory required is

$$M = \frac{1}{P}\sum_{i=0}^{\log n} 2^i n_i^2 = \frac{1}{P}\sum_{i=0}^{\log n} 2^i \left(\sqrt{\frac{n}{2^i}}\right)^2 = \frac{n}{P}\log n \tag{6.4}$$

In the $3D$ case, assume that $P = P_{XY} \times P_z$, where P_z is the number of $2D$ grids of size $P_{XY} = P_x \times P_y$ and $P_z = 2^l$ for some integer l; thus, $l = \log P_z$.

The root node is replicated on all the process layers. Thus it requires $n \cdot 2^l$ memory. Similarly, if $i < l$, level-i will be replicated on 2^{l-i} grids and will require $n \cdot 2^{l-i}$ words. If instead $i > l$, there will be no replication as each subtree resides in only a single 2D grid. Therefore, for $i > l$, the LU factors will require n memory in each level. Thus,

total memory required can be written as:

$$M_{3\mathrm{D}}(n,\, P,\, P_z) = \frac{1}{P}\left(\sum_{i=0}^{l} n2^{l-i} + \sum_{i=l+1}^{\log n} n\right) \approx \frac{1}{P}\left(2nP_z + n\log\frac{n}{P_z}\right). \tag{6.5}$$

Per-process communication

2D sparse LU. From Equations (6.2) and (6.4), the per-process communication volume of the 2D algorithm on planar graphs is

$$W_{2\mathrm{D}} = \frac{n\log n}{\sqrt{P}}. \tag{6.6}$$

3D sparse LU. We separately calculate the communication in the SUPERLU_DIST_2D-step, W_{3D}^{xy}, from the Ancestor Reduction step, W_{3D}^{z}, of Algorithm 6.[1]

Communication required in factorization: For the factorization of supernodes in the etree levels $i > l$, each process grid works on a $1/2^l$ fraction of the matrix at level-i. Thus, the per-process communication is $\frac{n}{2^l\sqrt{P_{XY}}}$. However, in levels $i < l$, only 2^i processes participate. Thus, the per-process communication in this level is $\frac{n}{2^i\sqrt{P_{XY}}}$ for the processes participating in this level.[2] Thus, the total communication across the critical path is given by:

$$W_{3\mathrm{D}}^{xy}(n,\, P,\, P_z) = \sum_{i=0}^{l-1}\frac{1}{2^i}\frac{n}{\sqrt{P_{XY}}} + \sum_{i=l}^{\log n}\frac{1}{2^l}\frac{n}{\sqrt{P_{XY}}}$$

We can substitute $2^l = P_z$ and $P_{XY} = \frac{P}{P_z}$. Then, assuming that $\log n \gg l = \log P_z$,

[1] All communication in SUPERLU_DIST_2D occurs in the XY plane.

[2] Note that average per-process communication across all the processes will still be $\frac{n}{2^l\sqrt{P_{XY}}}$, but we are more interested in total communication in the critical path of the factorization.

this expression simplifies to

$$W_{3D}^{xy}(n,\ P,\ P_z) \approx \frac{n}{\sqrt{P}}\left(2\sqrt{P_z} + \frac{\log n}{\sqrt{P_z}}\right). \tag{6.7}$$

Equation (6.7) is per-process communication for any general 3D process grid. W_{3D}^{xy} has a minimum at

$$P_z = \frac{1}{2}\log n. \tag{6.8}$$

Thus, the minimum amount of communication is

$$W_{3D}^{xy}(n,\ P) \approx 2\sqrt{2}\frac{n}{\sqrt{P}}\sqrt{\log n}.$$

Per-process communication in ancestor reduction (W_{3D}^{z}): We calculate W_{3D}^{z} for grid-0 as it is the only grid that participates in all the levels. Grid-0 receives the root node, distributed among all P_{XY} processes, in each iteration of Algorithm 6. Then the combined per-process data it receives just for the root is $\frac{l \cdot n}{P_{XY}}$. This expression for the i-th level is $\frac{(l-i)\cdot n}{2^i P_{XY}}$. We sum this expression over all i to get per-process communication in the ancestor-reduction step along the critical path of 3D sparse LU, which is

$$W_{3D}^{z}(n,\ P,\ P_z) = \frac{nl}{P_{XY}} = n\frac{P_z \log P_z}{P}. \tag{6.9}$$

Total per-process communication on the critical path: The total per-process communication is $W_{3D} = W_{3D}^{z} + W_{3D}^{xy}$. From Equation (6.7) and Equation (6.9),

$$W_{3D}(n,\ P,\ P_z) = \frac{n}{\sqrt{P}}\left(2\sqrt{P_z} + \frac{\log n}{\sqrt{P_z}}\right) + n\frac{P_z \log P_z}{P}.$$

When we choose P_z by Equation (6.8), this becomes

$$W_{3D}(n,\ P,\ P_z) = \mathcal{O}\left(\frac{n\sqrt{\log n}}{\sqrt{P}} + n\frac{\log n \log\log n}{P}\right). \tag{6.10}$$

For any practical n, $P \gg \log n$ even for modest values of P. Thus, for fixed n the first term of Equation (6.10) dominates.

Latency

The latency for the 2D algorithm is $L = \mathcal{O}(n)$. The latency for 3D algorithm is the dimension of sparse matrix described by the local elimination tree of grid-0. For calculating latency of the 3D algorithm, we divide it into two parts: a) the latency of leaf subtree factorization; and b) the latency of ancestor tree factorization. In the case of planar matrices, most of the supernodes are concentrated in the leaf level subtrees. Therefore, the leaf subtree on grid-0 will have a dimension of $\mathcal{O}\left(\frac{n}{P_z}\right)$. On the other hand, the root of etree has a dimension of $\sqrt{n}$, and the node in the k-th level has a dimension $\sqrt{\frac{n}{2^k}}$. Therefore, the latency of ancestor tree factorization is given by :

$$L^A_{3\mathrm{D}}(n,P,P_z) = \sum_0^l \sqrt{\frac{n}{2^i}} < \sum_0^\infty \sqrt{\frac{n}{2^i}} = \sqrt{n}\frac{\sqrt{2}}{\sqrt{2}-1} = \mathcal{O}\left(\sqrt{n}\right)$$

Thus, total latency in the 3D case is:

$$L_{3\mathrm{D}}(n,\ P,\ P_z) = \mathcal{O}\left(\frac{n}{P_z} + \sqrt{n}\right) \tag{6.11}$$

When $P_z = \mathcal{O}(\log n)$, $L_{3\mathrm{D}}$ is a $\log n$ factor smaller than L. For minimizing the $L_{3\mathrm{D}}$, we should have $P_z = \mathcal{O}\left(\sqrt{n}\right)$, in which case $L_{3\mathrm{D}}$ is $\mathcal{O}\left(\sqrt{n}\right)$.

6.4.3 Non-planar input graphs

For the non-planar sparse matrices with a strongly 3D geometry, the 3D factorization algorithm can, in a hypothetical best case, reduce the communication volume only by a constant factor of $\frac{2^{4/3}-1}{2^{4/3}-2} = 2.9$ and latency by a factor of $n^{1/3}$.[3] For such matrices, the dimension of the separator is $n^{2/3}$. Asymptotically, the size of LU factors of the

[3] In the assumed hypothetical best case, we have infinite number of 2D grids and ignore the cost of reduction.

matrix is $\mathcal{O}\left(n^{4/3}\right)$ and almost 20% of it is concentrated in the top separator. So the 3D algorithm cannot reduce the asymptotic complexity of the algorithm. A large separator also means that replicating the top-level root among many 2D grids will rapidly increase the additional per-process memory. However, it can still reduce the communication and latency of the factorization. For interested readers, we give the expressions for memory, communication, and latency in Table 6.2.

6.5 Results

We evaluate 3D sparse LU against the baseline 2D algorithm. The main results show performance gains from the 3D algorithm at both small and large core counts on a variety of sparse matrices taken from real applications. In addition, we estimate the scaling limits of the 3D algorithm. Beyond measured performance, we quantify the effects of the 3D algorithm on the communication volume and memory usage.

6.5.1 Setup

We use SUPERLU_DIST's default parameters in our experiments. We ran our experiments on Edison cluster at NERSC. Each node of the Edison contains dual-socket 12-core Intel Ivy Bridge processors. We chose 4 OpenMP threads per MPI process after trying various MPI×OpenMP configurations for different test matrices on 16 nodes. The code was compiled with the Intel C compiler version 18.0.0 and linked with Intel MKL version 2017.2.174 for BLAS operations. We use the same parameters for 3D that we obtained by tuning the 2D code.

Test matrices

We used four planar and six non-planar matrices, summarized in Table 6.3. The planar matrices come from the discretization of two-dimensional PDEs (K2D5pt4096, S2D9pt3072) and circuit analysis (g3_circuit, ecology1). Five of the six non-planar ma-

Table 6.3: Test sparse matrices used in experiments

Name	Application	n	$\frac{nnz}{n}$	#Flop†	$T^{\ddagger}_{\text{fact}}$
audikw_1	Structural	9.4e+5	82.0	1.17e+13	5.70
CoupCons3D	Structural	4.2e+5	53.6	9.09e+11	1.10
dielFilterV3real	FEM/EM	1.1e+6	81.0	2.00e+12	3.80
ldoor	Structural	9.5e+5	44.6	1.69e+11	1.97
nlpkkt80	KKT matrices	1.1e+6	26.5	3.14e+13	10.48
G3_circuit	Circuit Sim.	1.6e+6	4.8	1.21e+11	3.33
Ecology1	Ecology/Circuit	1.0e+6	5.0	4.49e+10	1.36
K2D5pt4096	PDE	1.6e+7	5.0	3.26e+12	59.81
S2D9pt3072	PDE	9.4e+6	9.0	2.47e+12	26.02
Serena	Structural	1.4e+6	46.1	5.97e+13	19.49

† #Floating point operations in the baseline SUPERLU_DIST (dSparseLU2D)
‡ Factorization time in seconds for baseline algorithm on 16 nodes.

trices are from the discretization of 3D PDEs and one, matrix nlpkkt80, comes from non-linear optimization. The factorization time of the test matrices ranges from 10-55 seconds on 16 nodes when using the baseline 2D SUPERLU_DIST.

6.5.2 Performance of the 3D algorithm on 16 nodes

The 3D sparse LU configurations achieve 2-11.6× and 0.33-4.9× speedup[4] with respect to 2D SUPERLU_DIST for planar and non-planar matrices, respectively. The results appear in Figure 6.5, which shows the factorization time normalized by the baseline 2D SUPERLU_DIST for each matrix and process configuration. Columns correspond to different 3D process configurations. The leftmost column, $P_z = 1$, is the 2D algorithm; subsequent columns correspond to P_z values of 2, 4, 8, and 16. The factorization time is divided into two components, T_{scu} and T_{comm}. The T_{scu} is the time spent in Schur-complement update on the critical path of the 3D factorization and T_{comm} is the non-overlapped communication and synchronization time.

[4]Speedups of less than 1 are, of course, slowdowns.

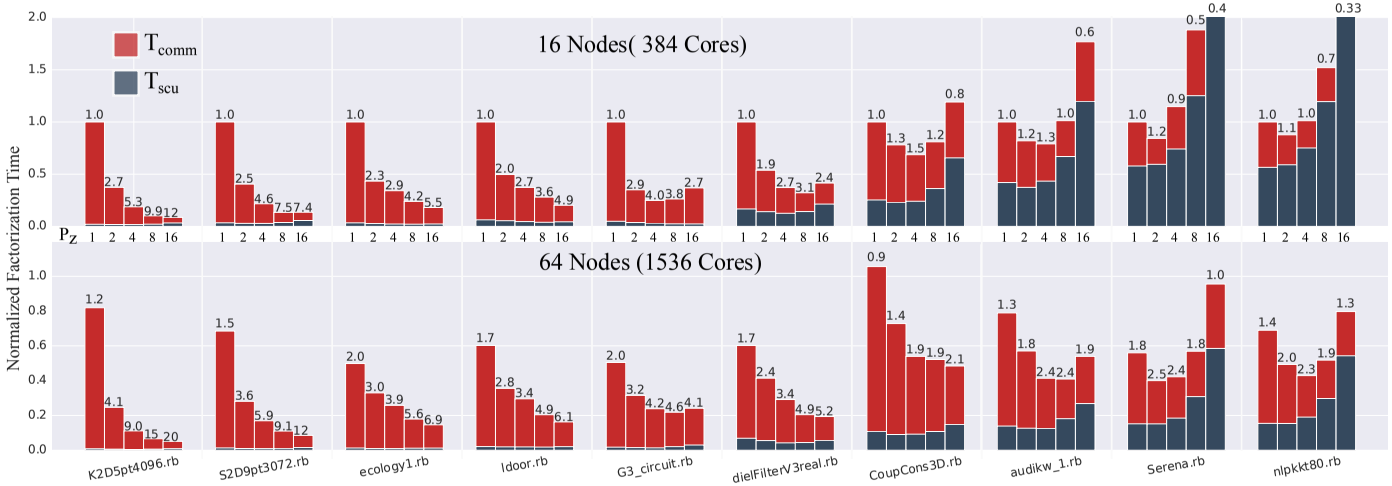


Figure 6.5: Performance comparison of various $P_x \times P_y \times P_z$ grids on 16 nodes (upper plot) and 64 nodes (lower) on the Edison system at NERSC. For each matrix, each column represents a different value of $P_z = 1, 2, 4, 8, 16$ from left to right. Thus, the leftmost column is the $2D$ algorithm, and when moving right, the $2D$ grids become smaller as P_z increases. For each data set, the time shown is normalized with respect to $2D$ SUPERLU_DIST on 16 nodes. T_{scu} is the time spent in the Schur-complement update on the critical path, whereas T_{comm} is the non-overlapped time spent in communication and synchronization.

Planar input graphs see better performance when P_z is large and the 2D grid size is small. Planar matrices have already very high communication cost at 16 nodes. We can see that T_{comm} decreases as we increase P_z. The profiling of K2d5pt4096 for the 2D algorithm shows severe load imbalance, which also has a cascading effect on the synchronization time. The 3D algorithm at $P_z = 2$ shows smaller time spent at synchronization points as it has roughly half synchronization point as the 2D algorithm. Some 3D matrices also achieve better performance when P_z is large and 2D grid size is small. For instance, ldoor comes from finite element discretization of a large door using a tetrahedral mesh. A "large door" is a very thin, or nearly planar, 3D object, and thus "partitions like a 2D object," meaning its top-level separator scales more closely to the way it would for a planar object rather than, say, a uniformly discretized 3D cube.

We also see a slowdown by up to 4× at $P_z = 16$ for extremely non-planar matrices Serena and nlpkkt80. For these, computation is still a large fraction of factorization time for the baseline 2D algorithm at 384 cores. Most of those computations are concentrated in the top few levels of the etree; therefore, reducing the 2D process grid size increases T_{scu}, which masks any gains from reduced communication.

6.5.3 Results on 64 Nodes

On 64 nodes, the 3D sparse LU configurations achieve 2-16.6× and 1.0-3.6× speedup with respect to 2D SUPERLU_DIST for planar and non-planar matrices, respectively. On 64 nodes the factorization time is qualitatively similar to the 16 nodes. However, for all the matrices T_{comm} dominates the factorization time for the baseline 2D algorithm. Therefore, even for extremely non-planar matrices Serena and nlpkkt80, 3D configurations achieve speedups of 1.7 and 1.9×, respectively.

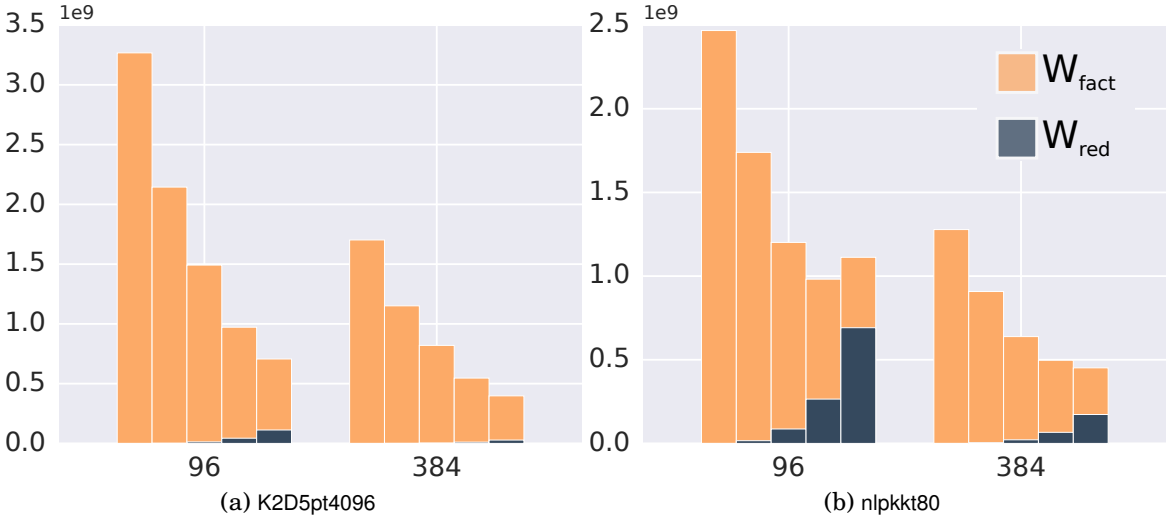


Figure 6.6: Per-process communication volume (in bytes) for different process grid configurations on 16 and 64 nodes (or 96 and 384 MPI processes, respectively). Each column in a group represents a $P_{XY} \times P_z$ process grid configuration, where $P_z \in 1, 2, 4, 8, 16$ from left to right (leftmost being a purely $2D$ configuration). Here, W_{fact} is number of words sent during the local factorization along the $2D$ grid, whereas W_{red} is number of words sent in the ancestor-reduction step along the z-axis.

6.5.4 Effects on per-process communication

For the planar graphs, 3D algorithm reduces per-process communication by 3-4.6× on 16 nodes and by 4-4.7× on 64 nodes. For non-planar graphs, 3D algorithm reduces per-process communication by 2.5-3.2× on 16 nodes and by 2.9-3.7× on 64 nodes. Figure 6.6 shows the per-process communication volume along the critical path of the 3D algorithm, for 16 and 64 nodes, and a planar and a non-planar matrix. We distinguish the number of words sent during the 2D factorization step (W_{fact}) and that of ancestor reduction (W_{red}) of Algorithm 6.

Observe that W_{fact} decreases with increasing P_z. Yet at large P_z, W_{total} can increase. For instance, W_{total} increases for nlpkkt80 at 16 nodes when going from $P_z = 8$ to 16. It is so as W_{red} increases almost linearly with P_z. Yet for planar graphs, this increase isn't much as they have very small separators at the top level. We estimate that for K2d5pt4096, W_{total} will increase with P_z after $P_z > 64$ at 96 processes.

Nevertheless, W_{red} decreases as $1/P_{XY}$ and W_{fact} does decrease as $1/\sqrt{P_{XY}}$ with

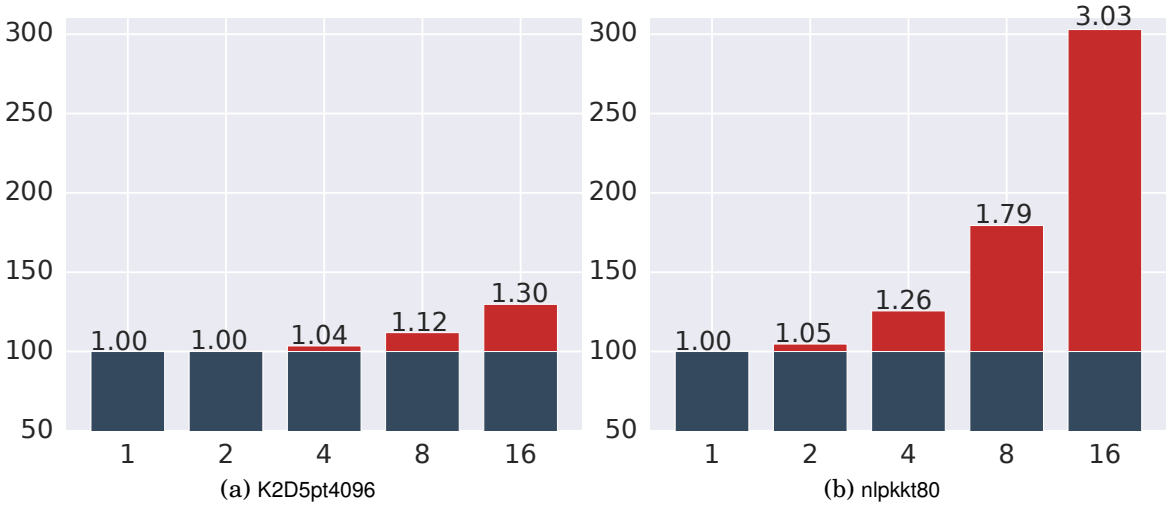


Figure 6.7: The relative memory overhead of 3D sparse LU algorithm over 2D (in percent).

increasing P_{XY}. So for larger P_{XY}, the cross-over P_z will be even larger.

6.5.5 Memory overhead

The 3D algorithm needs 30% more memory for the planar graph K2D5pt4096 and 200% more for the non-planar graph nlpkkt80, at $P_z = 16$ (see Figure 6.7). Memory overhead comes from replicating the dense separator blocks on all the process grids. Since the planar graphs have small separators, the memory overhead grows slowly with increasing P_z. Our model suggests that $P_z = \mathcal{O}(\log n)$ before memory overhead becomes comparable to memory of the LU factors. But non-planar graphs, like nlpkkt80, do not have good separators. Therefore, the memory overhead increases quickly. At $P_z = 16$, nlpkkt80 already needs 200% more memory. Overall at $P_z = 16$, memory overhead ranged between 18% to 245% for all matrices we tried.

6.5.6 Performance at Large Number of Cores

The best case speedup for a matrix is the speedup of the best $P_{XY} \times P_Z$ configuration relative to the best possible 2D process configuration. The best case speedup is ob-

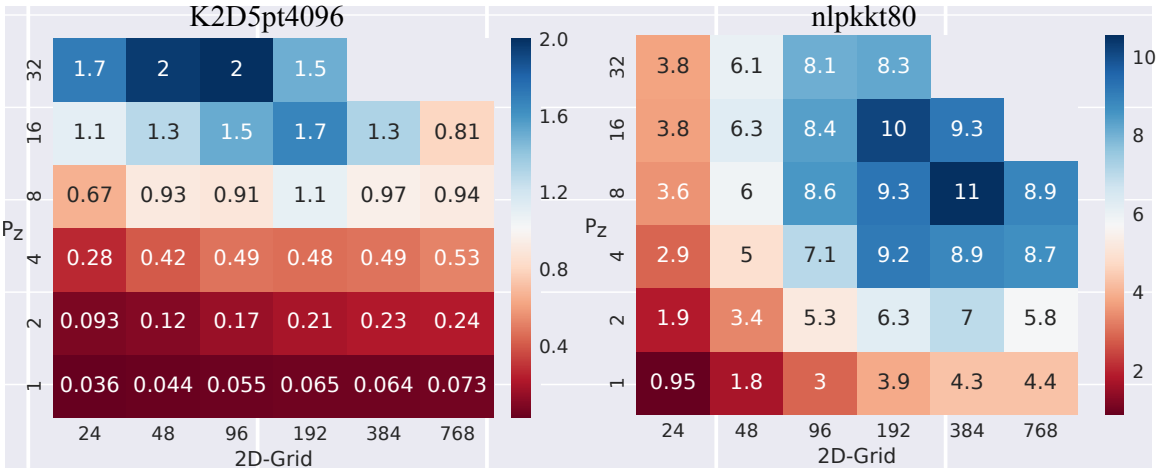


Figure 6.8: Performance (in TFLOP/s) of the 3D algorithm for different $P_{XY} \times P_z$ combinations. Here, we increase P_{XY} and P_z by a factor of 2 along x and y-axis respectively. Thus, bottommost row ($P_z = 1$) is the 2D algorithm.

served to be between 5-27.4× for planar graphs and 2.1-3.3× for non-planar graphs. We show a *heatmap* plot of performance for K2D5pt4096 and nlpkkt80 in Figure 6.8 for different combinations of $P_{XY} \times P_Z$. The performance is shown in trillions of floating-point operations per second (teraFLOP/s, or TFLOP/s).

Depending on their geometry and size, each matrix achieves its best performance at differing process grid configurations, $P_{XY} \times P_Z$. For a given $P = P_{XY} \times P_Z$, the planar graph K2D5pt4096 achieves the best performance along the line $P_{XY} = 24$. The strongly non-planar graph nlpkkt80 achieves its best performance along the line $P_z = P_{XY}/24$(↗) for a constant $P = P_{XY} \times P_z$. All of the remaining matrices achieved their best performance between the two lines.[5] In the best case, we achieved 27.4× speed-up for the graph K2D5pt4096. And on average the best 3D configuration was 6.5× faster than the best 2D configuration among all the matrices.

[5]If we had a completely dense matrix the best performance would have occurred along the line $P_z = 1$.

6.6 Related Work

The idea of using data replication to reduce communication in LU factorization goes back to Ashcraft, who described the first dense LU factorization based on a three-dimensional logical partitioning of the grid [39]. Ashcraft later presented the *fan-both* family of Cholesky factorization algorithm [40], which is a generalization of his 3D LU factorization algorithm. Later, Irony and Toledo [35] and Solomonik and Demmel [30] also described LU factorization algorithms using logical 3D partitionings of MPI processes. The central idea of all of the above work and ours is the same, namely, using multiple copies of the matrix to perform multiple Schur-complement updates in parallel. The total communication volume of all the above algorithms is $\mathcal{O}\left(n^2\sqrt[3]{P}\right)$, an asymptotic improvement over $\mathcal{O}\left(n^2\sqrt{P}\right)$ for 2D algorithms. However, these algorithms also increase the latency costs. Solomonik and Demmel showed that for such algorithms, communication volume costs are inversely proportional to the latency costs. Thus, despite the lower asymptotic communication complexity, the performance gains of these algorithms are limited even on communication bound problems. In contrast, our 3D algorithm reduces both bandwidth and latency by using the elimination tree parallelism.

It is possible to use the communication-avoiding dense LU algorithms to factor the dense nodes at the top levels of the etree. But we should try to avoid using them at the lower levels because of their increased latency costs.

Hulbert and Zmijewski [41] presented a column-oriented distributed sparse Cholesky. It can be considered as a special case of our 3D algorithm with $P_{XY} = 1$. For planar graphs, they achieve per-process communication volume costs of $\mathcal{O}(n \log P)$, as opposed to $\mathcal{O}\left(\frac{n\sqrt{\log n}}{\sqrt{P}}\right)$ as in our case.[6] However, their approach can only use $\mathcal{O}(\log n)$ processes for planar problems as opposed to $\mathcal{O}(n \log n)$ processes in the our 3D sparse LU algorithm. For sparse matrices with non-planar associated graph, $P_{XY} = 1$ will

[6] Indeed, we get the same expression if we substitute $P = P_z$ in Equation (6.9).

be extremely inefficient.

Multifrontal methods also use additional data to improve locality and communication. A notable such example is from Gupta et al. [42]. The per-process communication volume in their multifrontal sparse Cholesky algorithm for planar graphs is asymptotically $\mathcal{O}\left(\frac{n}{\sqrt{P}}\right)$, which is lower than the $\mathcal{O}\left(\frac{n}{\sqrt{P_{XY}}}\right)$ of our 3D algorithm by a factor of $\sqrt{P_z} = \sqrt{\log n}$ (see Table 6.1). However, their algorithm can use only $\mathcal{O}(n)$ processes in comparison to $\mathcal{O}(n \log n)$ processes for our 3D algorithm. Consequently, for achieving the same parallel efficiency, the per-process memory requirement for their algorithm increases with increasing n, whereas it remains constant for the 3D sparse LU algorithm. To achieve a similar parallel efficiency among their algorithm and ours, theirs will need $\mathcal{O}(\log n)$ more memory than our algorithm and reduce communication W by a factor of $\mathcal{O}\left(\sqrt{\log n}\right)$ to our algorithm. Thus, for such a scenrio, the two algorithms have the same $WM^{1/2} = \mathcal{O}\left(\frac{n^{3/2}\log n}{P}\right)$. For matrix multiplication-like dense linear algebra algorithms, it is known that [30, 43, 44, 45]

$$W = \Omega\left(\frac{\text{\# Arithmetic Operations}}{\sqrt{M}}\right). \tag{6.12}$$

The number of arithmetic operations for sparse LU factorization for the planar graph is $\mathcal{O}\left(n^{3/2}/P\right)$. Thus, if Equation (6.12) holds also for sparse matrices then our 3D algorithm and Gupta's multifrontal method are not optimal by a factor of $\mathcal{O}(\log n)$. However, it is likely that Equation (6.12) is not the same for a sparse LU factorization algorithm: dense computations perform $\mathcal{O}\left(n^3\right)$ operations on n^2 data, whereas sparse LU factorization of planar graphs perform $\mathcal{O}\left(n^{3/2}\right)$ operations on $n \log n$ data. Establishing similar lower bounds for sparse LU factorization as Equation (6.12) warrants further investigation. In addition, the dynamic memory requirement of the multifrontal method can be prohibitive and does not scale well with increasing number of processors. That is, the per-process memory requirement may increase with increas-

ing numbers of processors. As such, there has been a significant effort to improve the memory scalability of such methods [46, 47]. In other words, multifrontal methods trade off parallelism and performance with memory requirements.

Similarly to the multifrontal method, our 3D algorithm also uses elimination tree parallelism to reduce communication. Our mapping of subtrees to process layers is very similar to tree-based mapping algorithms for multifrontal methods. Also, our 3D LU factorization remains predominantly right-looking, which algorithmically is very different from the multifrontal methods. A comprehensive discussion on differences in right-looking and multifrontal methods can be found elsewhere [7, 8].

The use of the elimination tree parallelism to improve the scalability of the right-looking direct solver has also been explored previously, albeit, with a focus on hiding communication by pipelining and overlapping with computation. As such, it does not reduce communication volume [11].

Researchers have also proposed communication-avoiding pivoting strategies to make LU factorization more scalable [48, 49, 50]. Since SUPERLU_DIST uses static pivoting with iterative refinement, these techniques are not needed.

Among sparse direct solvers, prior work has studied efficient scheduling [51, 52, 53, 54, 55, 56, 11]. To improve the overlap of communication and computation, efficient lookahead techniques are part of state-of-practice for both dense and sparse direct solvers [57, 58, 11]. Lacoste [59] and Hugo [60] have also addressed memory and compute resource management for scaling multifrontal sparse direct solvers. The baseline SUPERLU_DIST incorporates similar techniques.

6.7 Conclusion

Our new 3D algorithm shows precisely how communication-avoiding techniques, namely the use of data replication as originally developed for dense LU, can be extended to the sparse case. Our discussion was limited to right-looking LU factor-

ization and static pivoting. However, we believe these principles could be applied to other variants of sparse factorization, such as Cholesky or QR decomposition.

In previous work, we proposed techniques for SUPERLU_DIST that can exploit manycore co-processors (e.g., GPU and Xeon Phi). Our "HALO" algorithm for accelerator offload can be seen as an instance of the 3D sparse LU algorithm presented in this paper where accelerators form another parallel 2D grid in addition to the host multicore CPUs. Despite that, HALO works much better for matrices that have large dense blocks as in case of very sparse matrix Schur-complement update cost is not dominant, which is the primary target of HALO's co-processor acceleration. On the other hand, the 3D sparse LU factorization performs better for relatively sparser matrices with small dense separators. We plan to add HALO to the 3D algorithm for hybrid clusters. We believe that by combining the two, we can potentially improve performance across a wider spectrum of matrices and platforms.

To improve the performance of the 3D algorithm for matrices with large dense blocks, we can in principle use a dense 2.5D LU algorithm to factor the supernodes on levels where we currently only use a subset of 2D grids. Alternatively, for those levels, we can merge two 2D grids to make a larger 2D grid to factor denser blocks. However, doing so would require significant changes to the data structure. Consequently, we leave this idea for future work.

CHAPTER 7

FAULTS IN COMPUTING SYSTEMS

7.1 Introduction

Informally, a fault is any occurrence of hardware deviating from its expected behavior. We usually view computing machines as reliable devices, without having to consider the impacts of random faults during the execution of the algorithm. But as the number of components in a computing system increases, such faults may become a norm rather than an exception. Moreover, hardware faults present a new threat to the overall scalability of algorithms, as the reliability of computing decreases with increasing number of concurrently active computing elements.

In this chapter, we will present a brief overview of faults in computing systems, their sources and classification, and mitigation techniques. Our main contribution is algorithmic techniques for making algorithms fault- tolerant discussed in Chapter 8. This chapter covers the necessary background and related work to understand our new methods.

7.2 Source of Faults

The challenges to reliability posed by continued decreases in CMOS digital circuit feature sizes motivate this work. Many analysts predict increases in, for instance, bit flips in cache and incorrect output from floating point units. Moreover, these error rates increase linearly with number of processors, with the number of cores on large-scale systems today numbering in the hundreds of thousands to millions [61]. On IBM's BlueGene/Q, observers report that today transient errors may occur in the L1 cache every 4 to 6 hours [62].

In hardware, ECC safeguards against bit flips in memory [63]. However, this still leaves caches, registers, and execution units vulnerable. In such cases, triple modular redundancy at the hardware level is possible [64, 65]. However, this technique is thus far not yet considered sufficiently mature for general-purpose computing.

7.3 Classification of Faults

We consider a *fault* to include any instance in which the underlying hardware deviates from its intended behavior. We will use the taxonomy of faults outlined by Hoemmen and Heroux [66], among others, as summarized below.

We distinguish *hard faults* and *soft faults*. A hard fault interrupts the program, causing it to immediately crash or terminate. This occurs when, for instance, a node or network link fails. By contrast, a soft fault does not cause immediate interruption of the program. L1 cache bit flips are a common type of soft fault.

Soft faults can be further divided into *transient* or *non-transient* faults. A transient fault is temporary. Examples include temporarily incorrect output from the floating point unit or a momentary bit flip while reading data from memory. By contrast, non-transient faults are permanent. For instance, suppose the input data stored on disk is corrupted. In this case, reading the data may succeed but the data is wrong on every read.

Our work concerns only transient soft faults. Thus, in the consequent, a "fault" is a transient soft fault unless otherwise noted.

7.3.1 Failure

We term failure an event where a fault causes the output to fall outside acceptable limits or algorithm fails to terminate. Not all faults may lead to a failure. Typically, a soft-fault may cause an error in computation or corruption in data. This error or corruption may propagate to other parts of the calculation and eventually lead to

incorrect results, thus a failure in execution.

7.3.2 Silent Data Corruption

Silent Data Corruption (SDC) is a particularly insidious manifestation of soft-faults. A soft-fault may cause a data corruption that propagates and corrupts entire the data without any notification to the program or operating system. Eventually, SDC causes the algorithm to fail without any warning of failure. Such occurrence of SDC will cause serious reliability issue in computing.

7.4 Model of Reliability

For an algorithm to terminate successfully, at least some amount of computation must be done reliably. However, in general we do not have control over which operations are done correctly and which operations are done incorrectly. Here, we will attempt to distinguish algorithmic operations that *must* be performed reliably from those that may be performed unreliably. We refer to these different modes of computation as *reliable mode* (or *reliable computation*) from *unreliable mode* (or *computation*), without saying precisely how to implement these modes. The prior work of others similarly assumes precisely this form of *selective reliability* [66].

Having said that, proposals exist for implementing reliable mode, both in software and in hardware. At the software level, it can be achieved using by redoing some of the computation using process replication [67] or triple modular redundancy. When running in unreliable mode, we do not know whether the computation has executed correctly or not. Therefore, to carry out an analysis, we will necessarily assume some model of faults when running in unreliable mode.

7.5 Conclusion

In the next generation computing systems, hardware faults will pose a new challenge to the scalability of algorithms. We expect that a coordinated effort from hardware, system software, and algorithmic resilience would be necessary to overcome impacts of faults. In this thesis, our focus is on algorithmic techniques to improve the resilience of the solver algorithms. In Chapter 8, we introduce the principle of self-stabilization for constructing algorithmically resilient iterative linear solvers.

CHAPTER 8

SELF-STABILIZING ITERATIVE SOLVERS

8.1 Introduction

Informally, a system is *self-stabilizing* if it reaches a valid state within a finite number of steps, regardless of its initial state (valid or not). The self-stabilizing property implies the possibility of fault-tolerance: if a transient fault causes the system to enter an invalid state, then the self-stabilizing property ensures the system can eventually return to a valid state. The formal design of self-stabilizing systems, primarily for combinatorial algorithms in protocol design, dates back at least to the seminal work of Dijkstra (1973) [68]. It now appears in many settings, including distributed systems, network routing protocol design, and control theory, to name just a few [69]. Given the current interest in resilient numerical computing on extreme-scale machines, we are trying to apply self-stabilization to the design of iterative solver algorithms. Whether this idea can work is unknown, but this chapter describes an initial "point result" suggesting promise.

Our basic recipe for applying self-stabilization to numerical algorithm design is roughly as follows. First, we regard the execution of a given solver algorithm as the "system." Its state consists of some or all of the variables that would be needed to continue its execution. Then, we identify a condition that must hold at each iteration for the solver to converge. When the condition holds, the solver is in a "valid" state; otherwise, it is in an invalid state. Finally, we augment the solver algorithm with an explicit mechanism that will bring it from an invalid state to a valid one within a guaranteed finite number of iterations. Under specific assumptions on the type and rates of transient faults (Section 7.3), we will show our modified solvers are self-

stabilizing.

Some iterative solvers are inherently self-stabilizing. These include globally convergent stationary iterations, e.g., Gauss-Seidel, Jacobi, and Newton iterations for nonlinear systems. But many others are not self-stabilizing in their standard formulations. Among these are the steepest descent and conjugate gradient methods for solving symmetric positive definite linear systems, which we consider here. Nevertheless, it may be possible to construct self-stabilizing versions thereof, which we show as a proof-of-concept (Section 8.2). We then offer both analytical and empirical evidence of their efficacy and their limitations (Section 8.3). Regarding the latter, our conclusions apply only to the subclass of transient soft faults in which numerical values are temporarily perturbed by bit flips at a specified rate.

Why would self-stabilization—if it can be achieved at all—be an attractive paradigm, relative to more established approaches? For instance, we might instead choose traditional checkpointing and algorithm-specific checksumming techniques. (See also Section 8.5.) The basic paradigm in these other methods is to save the state of the system periodically, to detect faults, and when faults occur to restore to a previously correct state. Checksums provide a way to detect when a fault occurs. Critically, these methods rely on accurate fault *detection*. If a fault goes undetected, it can propagate and lead to incorrect results, or in the worst case, terminate but report the result as correct [62]. Moreover, the cost of such methods can be prohibitively expensive at high fault rates, given the often high cost of checkpoint, restart, and perhaps also fault detection operations. And in the extreme case that there is some error at every iteration, these classical approaches will most likely fail to make any progress since there is never any correct state to save.

By contrast, self-stabilization can obviate the need for full state saving and fault detection. When a fault occurs, a self-stabilizing algorithm will by design reach a correct state in a finite number of steps. The valid state might not be identical to

some state of a fully fault-free execution; however, the self-stabilizing property ensures convergence, in the absence of additional faults. In the extreme case of some error at every iteration, we show that a self-stabilized iterative solver can still make progress, albeit at a slower rate.

8.2 Self-Stabilizing Iterations

This section describes and characterizes self-stabilizing iterative solver algorithms and some of their properties.

We regard the *state* of the (iterative solver) algorithm at a given point during its execution as a subset of its variable values that is sufficient to restore and resume its execution at that point. The choice of state variables may be a matter of convenience; for instance, a particular choice of variables may have some redundancy in order to reduce the amount of computation needed to restore execution.

We say the algorithm is in a *valid state* if, under a fault-free execution of the algorithm from that state, the algorithm still produces a correct result. In the case of an iterative solver, correctness is usually relative to some convergence criterion. As such, the algorithm is in an *invalid state* if in its subsequent execution from that state it either 1. does not converge or otherwise aborts on an error condition; or 2. converges to an incorrect value. In general, these are non-trivial properties to check.

By its design, an algorithm that starts from some valid state will, in a fault-free execution, remain in a valid state. A fault may cause a transition into an invalid state. A fault for which the algorithm either remains in a valid state or reaches a valid state after a finite number of iterations is a *tolerable* fault; otherwise, it is *intolerable*. The solution may, in the presence of tolerable faults, differ from a purely fault-free execution; and an intolerable fault will cause the algorithm to fail. Importantly, the concept of self-stabilization of an algorithm should be described only

with respect to class of faults, which motivates the following definition.

Definition 1. Self-stabilizing iterative algorithm. An iterative solver algorithm $\mathscr{A}$ is *self-stabilizing* with respect to class of faults $\mathscr{F}$ if, when a fault $f \in \mathscr{F}$ occurs, the algorithm either remains in a valid state or comes to a valid state in after finite number of iterations.

Under certain conditions, the self-stabilizing property of definition 1 is equivalent to fault tolerance. When f occurs, a self-stabilizing iterative solver algorithm will return to a valid state after finite number of iterations, and thereby eventually converges. Note that self-stabilization is a weak form of fault-tolerance: it only says the algorithm can reach a valid state, but does not guarantee convergence. This is an inherent limitation of the approach. However, many algorithms have smoothing properties that, when combined with the self-stabilization, may result in robust fault tolerant algorithms.

Some of the algorithms are inherently self stabilizing with respect to transient soft faults including stationary iterations (Jacobi, Gauss-Seidel, Richardson) and some nonlinear iterative methods (fixed-point iteration, Newton iteration). **Naturally self-stabilizing methods.** A *naturally self-stabilizing* algorithm is self-stabilizing without modification. There are numerous examples. For example, consider the stationary iteration for solving $Ax = b$ whose form is

$$(A+B)x_{k+1} = Bx_k + b.$$

Under certain conditions on A, B, and b, this algorithm is globally convergent, meaning it will converge to the true solution for *any* starting value of x_0 [70]. If a fault occurs in iteration i, then it is possible $(A+B)x_{i+1} \neq Bx_i + b$. However, x_{i+1} is still a valid state because of the global convergence property of the algorithm.[1] Hence, the

[1]Other examples of similarly self-stabilizing algorithms owing to a global convergence property

algorithm is naturally self-stabilizing.

Periodic correction schemes. If the algorithm is not naturally self-stabilizing, the interesting research question is whether we can modify it to be so.

From the perspective of traditional algorithm-based fault-tolerance (ABFT) schemes, we can make an algorithm self-stabilizing by checkpointing a valid state, detecting a fault, and restarting at the most recent valid state when a fault is detected. If it is not possible to do efficient fault detection and checkpoint/restart, we may seek a different approach.

A natural idea is to introduce a *periodic correction step*. For a given algorithm, we seek a correction step such that when it executes, either (a) if the algorithm is in a valid state, it remains in a valid state; or (b) if the algorithm is in an invalid state, the correction moves it into a valid one. By applying the correction step after a fixed (bounded) number of iterations, we may effectively bound the number of iterations to reach a valid state. Thus, such a periodically corrected algorithm meets the self-stabilizing criteria of definition 1.

There are at least two more important issues in designing a correction step. The first is its cost. Since it is extra computation, we seek corrections that are cheap or may otherwise be applied at infrequent (but still fixed, per the finite steps requirement) intervals. The second is the reliability mode when executing the step. In general, we will assume the selective reliability model in which the correction step executes in guaranteed reliable mode. However, it is possible that for infrequent faults, this restriction may be relaxed.

To show corrections are possible, below we present two case studies in the context of solving $Ax = b$. They show how to transform the standard formulations of steepest descent (Section 8.2.1) and conjugate gradients (Section 8.2.2) methods for solving $Ax = b$ into self-stabilizing forms.

include other stationary iterations (Jacobi, Gauss-Seidel, Richardson) and some nonlinear iterative methods (fixed-point iteration, Newton iteration).

8.2.1 Self-stabilizing Steepest Descent

We show the standard form of steepest descent in Algorithm 8. In this method, the solution of the N-dimensional system, $Ax = b$ is the solution to an optimization problem. Specifically, it seeks to minimize the N-dimensional paraboloid, $F(x) = \frac{1}{2}x^T Ax - x^T b$. In each step, the steepest descent algorithm finds the direction of maximum descent along F. When A is symmetric positive definite (SPD), this direction at iteration k is the residual, $r_k = b - Ax_k$, where x_k is the current approximate solution; the algorithm calculates a step length α in the direction of r_k so as to minimize $F(x_k + \alpha r_k)$ and computes a new estimate $x_{k+1} \leftarrow x_k + \alpha r_k$.

We may represent the state of Algorithm 8 at iteration k by (x_k, r_k). In a fault-free execution, the state variables satisfy

$$r_k = b - Ax_k. \tag{8.1}$$

One can show Algorithm 8 only converges to x when Equation (8.1) holds.

If Equation (8.1) does not hold, then perturbation theory tells us that the algorithm solves a different problem, $Ax = \tilde{b}$, where $\tilde{b} = r_k + Ax_k$. Hence, $(x_k,\ r_k)$ is a valid state if

$$\|r_k - b + Ax_k\|_2 < \epsilon_1 \|r_0\|_2 \tag{8.2}$$

where ϵ_1 determines our acceptable convergence criterion, as given in line 3 of Algorithm 8.

Algorithm 8 is *not* self-stabilizing. In particular, suppose a fault occurs at line 4. Then, Equation (8.1) no longer holds. If the fault is such that Equation (8.2) is still satisfied, the algorithm remains in a valid state. However, it is also possible the algorithm will instead reach an invalid state and will continue to remain in an invalid state. Thus, the algorithm will fail.

Algorithm 8 Steepest descent for solving the symmetric positive definite system, $Ax = b$

Require: A, b, x_0 (Initial guess), and ϵ (threshold)

1: i=0 ; $r_0 = b - Ax_0$; $x_i = x_0$;
2: $\|r_i\|^2 = \|r_0\|^2 = r_i^T r_i$;
3: **while** $||r_i|| > \epsilon_1 \times \|r_0\|$ **do**
4: $q_i = Ar_i$;
5: $\alpha_i = \|r_i\|^2/(r_i^T q_i)$;
6: $x_{i+1} = x_i + \alpha_i r_i$;
7: $r_{i+1} = r_i - \alpha_i q_i$;
8: $\|r_{i+1}\|^2 = r_{i+1}^T r_{i+1}$;
9: $i = i + 1$;
10: **return** x_i

To avoid reaching an invalid state, we need to restore the relation of Equation (8.1). This can be done by *forcing* r_k to be equal to $b - Ax_k$, as shown by the correction step on line 5 of the modified algorithm, Algorithm 9.

This correction costs one extra matrix-vector product *in reliable mode*. Assuming line 4 of Algorithm 8 is the most expensive, the correction can cost much more than double the cost if reliable mode is costs much more than unreliable mode. As such, we guard the correction step as line 4 of Algorithm 9 indicates. This test ensures that we execute the correction only once every F iterations, for some fixed F. This frequency is a tuning parameter that depends on the largest possible error of the fault and the condition number of A.[2]

Since line 5 of Algorithm 9 enforces the validity condition of Equation (8.1) and since the F parameter of line 4 provides a way to bound the number of iterations to move from an invalid to valid state, we may conclude the algorithm is self-stabilizing.

8.2.2 Self-stabilizing Conjugate Gradient

Conjugate gradient (CG) is another descent-type algorithm for SPD systems. In principle, CG converges faster than steepest descent; and in exact arithmetic, CG con-

[2] If the error of the fault can be bounded, we may be able to estimate F. For instance, rounding errors alone suggest correcting every $\mathcal{O}(\sqrt{N})$ iterations [71]. Thus, we should expect to correct more frequently than that.

Algorithm 9 Self-stabilizing steepest descent. The parameter F is the correction step frequency in iterations.

Require: A, b, and x_0 (Initial guess), F, ϵ

i=0; $r_i = b - Ax_0$; $x_i = x_0$
$\|r_i\|^2 = \|r_0\|^2 = r^T r$;
while $||r_i|| > \epsilon \times r_0$ **do**
    if $i \equiv 0 \pmod{F}$ **then** ***Do Reliably***
        $r_i = b - Ax_i$
    $q_i = Ar_i$;
    $\alpha_i = \|r_i\|^2/(r_i^T q_i)$;
    $x_{i+1} = x_i + \alpha_i r_i$;
    $r_{i+1} = r_i - \alpha_i q_i$;
    $\|r_{i+1}\|^2 = r_{i+1}^T r_{i+1}$;
    $i = i + 1$;
return x_i

Algorithm 10 The conjugate gradient (CG) algorithm for the SPD system, $Ax = b$

Require: A, b, x_0 (initial guess), ϵ, and M (max iterations)

$r_0 = b - Ar_0$; $p_0 = r_0$; $i = 0$
while $\|r_i\| > \epsilon \cdot \|r_0\|$ and $i < M$ **do**
    $q_i = A \times p_i$;
    $\alpha_i = \|r_i\|^2/(p_i^T q)$;
    $x_{i+1} = x_i + \alpha \times p_i$;
    $r_{i+1} = r_i - \alpha q$;
    $\beta = \|r_{i+1}\|^2/\|r_i\|^2$;
    $p_{i+1} = r_{i+1} + \beta p_i$;
    $i = i + 1$;
return x

verges in a finite number of iterations. CG is also specific to the SPD case, making it more storage-efficient than, for instance, GMRES [72] for general systems. A typical formulation of CG appears in Algorithm 10.

The CG method maintains a search direction p_k. Here r_k is the residual and p_k is the new search direction, given by

$$p_{k+1} = r_{k+1} + \beta p_k, \quad p_0 = r_0. \tag{8.3}$$

The coefficient β is chosen so that $p_{k+1}^T A p_k = 0$ corresponding to line number (7) in Algorithm 10. In each step x_k advances by $\alpha_k p_k$, where α_k is chosen so that it

minimizes $F(x_k + \alpha_k p_k)$ corresponding to line number (4) in the Algorithm (3).

In a fault-free execution of Algorithm 10, three global orthogonality relations hold:

$$\begin{aligned} p_i^T A p_j &= 0 \quad \text{if } i \neq j; \\ r_i^T r_j &= 0 \quad \text{if } i \neq j; \text{ and} \\ r_i^T p_j &= 0 \quad \text{if } i > j. \end{aligned} \tag{8.4}$$

To restore a particular state in which the relations of Equation (8.4) hold, one would have to store all previous search direction and residuals, which is not possible since storing them would require a lot of additional storage. However, in practice one can quickly lose these relations even under finite-precision arithmetic. Since we are interested in restoring the algorithm to a state which can still converge to a correct solution, we can relax some of these conditions.

In particular, we may invoke the following result from nonlinear CG [73]:

Theorem 1. *Starting from any state* (x_0, r_0, p_0) *such that* $r_0 = b - Ax_0$ *and* $r_0^T p_0 \neq 0$, *CG* converges *if the following relations are maintained for* $k \geq 0$*: 1.* $r_k = b - Ax_k$*; 2.* $\left(p_k^T r_k\right) / \|p_k\| \|r_k\| > c_1$ *for some* c_1 *if* $||r_k|| \neq 0$*; and 3.* α_k *is chosen to minimize* $F(x_k + \alpha p_k)$*, i.e.,* $\alpha_k = r_k^T p_k / p_k^T A p_k$.

Theorem 1 is a corollary to the Zoutendijk condition, which states that, if conditon (3) of theorem 1 holds then so, too, does [73]

$$\sum_{i=0}^{\infty} \left(\frac{\left(p_k^T r_k\right)^2}{\|p_k\|^2 \|r_k\|^2} \right) \|r_k\|^2 < \infty. \tag{8.5}$$

Condition (1) of theorem 1 ensures the correctness of the problem that we are trying to solve. Condition (2) of theorem 1 ensures that until algorithm has converged ($\|r_k\| = 0$), there is a corresponding search direction p_k which also satisfies condition (2).

Equation (8.5) says that the series $\frac{\left(p_k^T r_k\right)^2}{\|p_k\|^2 \|r_k\|^2} \|r_k\|^2$ is convergent, hence

$$\lim_{k\to\infty} \frac{\left(p_k^T r_k\right)^2}{\|p_k\|^2 \|r_k\|^2} \|r_k\|^2 = 0$$

Since condition (2) ensures that $\left(\left(p_k^T r_k\right)^2 / \|p_k\|^2 \|r_k\|^2\right)$ is bounded from below by c_1, then convergence of the series would entail $\lim_{k\to\infty} \|r_k\|^2 = \|b - Ax_k\| = 0$. Hence, if the conditions of theorem 1 are satisfied then algorithm would converge to the correct solution.

Note that theorem 1 is just one possible set of sufficient conditions that one can use to construct a correction step.

Suppose we choose to restore these conditions. Then, we may choose the quadruplet $(x_k, r_k, p_k, \alpha_k)$ as the state variable and construct the correction step as follows.

First, restore the residual condition by calculating $r_k = b - Ax_k$ explicitly and check for convergence.

Next, check the validity of condition (2) in theorem 1. If r_k and p_k are approximately orthogonal, as occurs when $r_k^T p_k < \epsilon_1 \|r_k\| \|p_k\|$, then we set $p_k = r_k$.

Then, choose α_k to minimize $F(x_k + \alpha_k p_k)$, as occurs if

$$\alpha_k = r_k^T p_k / p_k^T A p_k. \tag{8.6}$$

In a fault-free execution, Equation (8.6) and line 4 of Algorithm 10 are equivalent. However, if a fault does occur they are no longer equivalent, hence, expression in line 4 of Algorithm 10 no longer has the optimality property of Equation (8.6), thereby violating condition (3) of theorem 1.

Lastly, we have several choices for updating p_k. We choose to do so by a particular

Algorithm 11 Self-stabilizing conjugate gradient (SS-CG)

Require: A, b, x_0 (initial guess), ϵ, M, and F (correction frequency)

$r_0 = b - Ar_0$; $x = x_0$; $p_0 = r_0$; $i = 0$
while $\|r_i\| > \epsilon \cdot \|r_0\|$ and $i < M$ **do**
  if $i \equiv 0 \pmod{F}$ **then** ***Do Reliably***
    $[r_i, q] = A \times [x_i, p_i]$ ▷ May read A just once
    $r_i = b - r_i$
    $\alpha_i = r_i^T p_i / (p_i^T q)$;
    $x_{i+1} = x_i + \alpha \times p_i$;
    $r_{i+1} = r_i - \alpha q$;
    $\beta = -r_{i+1}^T q_i / (p_i^T q_i)$;
    $p_{i+1} = r_{i+1} + \beta p_i$;
  else
    $q_i = A \times p_i$;
    $\alpha_i = \|r_i\|^2 / (p_i^T q_i)$;
    $x_{i+1} = x_i + \alpha \times p_i$;
    $r_{i+1} = r_i - \alpha q$;
    $\beta = \|r_{i+1}\|^2 / \|r_i\|^2$;
    $p_{i+1} = r_{i+1} + \beta p_i$;
  $i = i + 1$;
return x

choice of β that enforces A-orthogonality of p_k and p_{k+1}:

$$p_{k+1} = r_{k+1} + \beta p_k, \quad \beta = -p_k^T A r_{k+1} / p_k^T A p_k \tag{8.7}$$

We choose this form for two reasons. First, it will guarantee that our overall self-stabilizing version of CG is mathematically equivalent to CG during a fault-free execution. Secondly, we want to be able to prove the self-stabilizing property holds and, moreover, maintain at least a few *local* orthogonality properties (see theorem 2 below).

Combining these four state variable correction steps together, the new and complete algorithm is Algorithm 11. Note that Algorithm 11 requires a fused matrix-vector product in reliable mode. However, in modern architecture cost of 2 fused matvecs are same as one [74]. Now we can state the following local orthogonality relations, which the correction step maintains.

Theorem 2. *Let Algorithm 11 start in any state,* (x_0, r_0, p_0)*. If a correction is performed in the* k^{th} *iteration and all subsequent iterations are fault-free, then the following* local orthogonality conditions *will be true (in exact arithmetic):*

$$\begin{array}{lcll} r_l & = & b - Ax_l & \text{if } l > k \\ p_{l+1}^T A p_l & = & 0 & \text{if } l \geq k \\ r_{l+1}^T r_l & = & 0 & \text{if } l > k \\ r_{l+1}^T p_l & = & 0 & \text{if } l \geq k \\ r_l^T A p_l & = & p_l^T A p_l & \text{if } l > k \\ r_l^T p_l & = & \|r_l\|^2 & \text{if } l > k \\ \left(p_l^T r_l\right)/\|p_l\|\,\|r_l\| & > & \lambda_1/\lambda_n & \text{if } l > k \end{array} \tag{8.8}$$

We need to show that Algorithm 11 is indeed self-stabilizing. We have already proven condition (1) of theorem 1. Equation (8.8) validates condition (2) of theorem 1. Lastly, the choice of $\alpha_l = r_l^T r_l / r_l^T A p_l = r_l^T p_l / p_l^T A p_l$ retains the minization condition (3) of theorem 1. Hence, by theorem 1, Algorithm 11 will converge to the solution of $Ax = b$.

In the presence of faults, self-stabilizing CG loses the global orthogonality properties of fault-free CG, and therefore also the finite termination property of Krylov subspace methods. However, it at least maintains local orthogonality and optimality properties as argued above. Note that these local relations hold in case of inexact Krylov subspace based methods [75]. While the finite termination property is in theory highly desirable, our experiments will show that good rates of convergence are nevertheless possible without it, even when fault rates are relatively high.

8.3 Numerical Experiments

We performed a series of numerical experiments in MATLAB to test the robustness of the self-stabilizing algorithms in Section 8.2. We focus on CG and omit results for

steepest descent.

8.3.1 Fault Injection Methodology

Since all steps of CG depend on the matrix-vector product, we emulate transient soft faults by injecting bit flips into this operation. In particular, at the beginning of each iteration of CG, we start with the uncorrupted matrix A, flip some of its bits in an independent and uniformly random way, perform the (possibly corrupted) matrix-vector product, and then restore the uncorrupted A (since we assume transient errors). Since a fault in the matrix-vector product propagates to most of CG's other variables, considering errors here is sufficient to corrupt the entire algorithm's results.

Our bit flipping procedure assumes independence between bits and iterations. Specifically, we model bit flips as Bernoulli trials, with each bit of the numerical values of A flipping independently with a uniform probability P. Depending on the value of P compared to the total number of bits in the matrix values, a given matrix-vector product may contain no bit flips in a given iteration when P is small or multiple bit flips in every iteration if P is large.

In each step we check for not-a-number (NaN) and infinity (Inf) values in the result of the matrix-vector product and in the residual r_i and replace those values with random numbers. This check is the only form of fault-detection that we use in our self-stabilizing algorithm.

Lastly, we assume reliable mode for the matrix-vector product of the correction step, as assumed by others (Section 7.3).

8.3.2 Solvers

Our evaluation compares several solver methods.

*Fully reliable CG (*ERROR FREE CG*).* This method is standard CG (Algorithm 10)

where all computation is done in reliable mode. That is, it assumes *no* faults; as such, it will generally require the fewest iterations to converge relative to other CG-type methods but has no built-in fault-tolerance.

*Restarted CG (*CG-RES*).* This method is a known variant of CG and might be a natural choice for fault-tolerance. The basic idea in CG-RES is simply to reset the search direction p_k periodically to be the steepest descent direction. The intuitive aim of doing so is to compensate for any error that may have accumulated in p_k. This approach also makes CG-RES a type of self-stabilizing algorithm, if we assume the restart step is done in reliable mode. However, CG-RES is *not* mathematically equivalent to standard CG. Restarting CG usually slows the rate of convergence. One may also view CG-RES as a type of checkpoint/restart method in which only x_k is checkpointed. (It is not possible to know that p_k is a correct direction.) As such, it is a useful baseline for comparison against our method.

*Self-stabilizing CG (*CG-SS*).* This is our algorithm of Algorithm 11. Recall that only the correction step (every $F = 10$ iterations) runs in reliable mode.

*Fault-tolerant GMRES (*FT-GMRES*).* This method is based on the inner-outer iteration paradigm [66]. Outer iterations use GMRES and run in reliable mode. Inner iterations run unreliably and may use any solver; we tried both standard CG and GMRES; we report results on each test problem using whichever yielded the fewest total iterations.

Regarding our use of FT-GMRES, we make two qualifying remarks. First, for SPD systems, GMRES and CG generally exhibit similar convergence rates in a fault-free execution. However, a practitioner will often prefer CG over GMRES because GMRES-type methods carry a significantly higher storage overhead. Secondly, FT-GMRES is really a *preconditioned* method, whereas we are comparing to CG methods (including our CG-SS) *without* preconditioning. Thus, when comparing convergence and iteration rates, one should use caution. Nevertheless, we include FT-

Name	N	NNZ	$\kappa(A)$
K3D	27000	183600	646
DIAG	10000	10000	990100
THERMAL1	82654	574458	496250

Table 8.1: Different problems used for experimentation

GMRES as it is the closest comparable method with built-in fault-tolerance of which we are aware.

8.3.3 Test problems

We choose 3 different test problems, summarized in Table 8.1.

The first test problem is K3D comes from finite difference discretization of the 3D poisson equation on a cubic domain of size $30 \times 30 \times 30$. This problem is well-conditioned and shows superlinear convergence with fault-free CG, even *without* preconditioning.

The second test problem is DIAG, a diagonal matrix of size $N = 10,000$ with each of its entries given by

$$D(i) = 100 + \sqrt{1 + 10^k (i-1)/N},$$

where N is the size of the matrix and k is a conditioning parameter. (We use $k = 16$ in Table 8.1.) To generate a right-hand side, we generate a random vector and multiply by it; note that this method allows us to measure absolute error. Furthermore, a diagonal test problem is useful because it permits easy explicit control of the spectrum, and thus convergence rates, by tuning k [76, 66]. Compared to K3D, DIAG is relatively ill-conditioned.

The third test problem is THERMAL1, which is #1402 from the University of Florida Sparse Matrix Collection [21]. The source application is steady-state thermal analysis using finite element analysis on an unstructured domain. The problem is relatively ill-conditioned and exhibits sub-linear convergence under ERROR FREE

CG; as such, we would usually want to precondition it. However, here we wish to stress-test the basic CG-SS approach and so we omit preconditioning.

8.3.4 Experiments

For each test problem, we ran two sets of experiments, each considering a variation on silent data corruption [62].

1. We allow bit flips to occur in any part of the floating-point value.
2. We allow bit flips only in the mantissa and in the sign bits, but *not* the exponent.

In the first case, bit flips in the exponent cause unbounded errors and will be much harder to tolerate. However, there may be efficient ways to detect these cases or to apply circuit-level selective reliability to exponent computations.

In light of possible special measures for exponent corruption, the second case of mantissa and sign bit flips is worth considering separately. We might expect this second case when using, for example, probabilistic circuits, where we might compute the mantissa using lower-voltage power-saving circuits at the cost of increased unreliability [77]. This subclass of faults is insidious in that it is harder to detect than exponent corruption; and even if detectable, it likely requires recomputation. However, a nice intellectual aspect of mantissa errors is that we may be able to bound the error analytically. We offer such an analysis in Section 8.4.

For the first case, we choose the bit-flip probability to yield an expected value of 4 bit flips in each unreliable matrix-vector product; and for the second case, 40 bit flips. These error rates are extremely high relative to estimates in the literature; however, it allows us to consider the case of high degrees of scaling, in which case problem size increases would also increase the number and frequency of bit flips.

8.3.5 Results

We compare the four solvers on each of the 3 test problems and two bit-flip scenarios. We separately assess convergence rates, measured in number of iterations to reach a given accuracy, and number of reliable matrix-vector products required.

Convergence Rate

Figure 8.1 compares the observed convergence history of the four algorithms. For FT-GMRES, we count iterations as cumulative of each inner iteration as a function of the number of unreliable matrix-vector products. Each column of plots is for a given test problem (K3D on left, THERMAL1 in the middle, and DIAG on the right), and each row considers a bit flip scenario (mantissa and sign bits only on the top, and all bits including the exponent on the bottom).

Consider first the left column of plots, in which only mantissa and sign bit flips occur. Recall the probability of bit flips is chosen so that the expected number of bit flips per unreliable matrix-vector product is 40 flips. This rate is relatively high and serves as a stress-test for all the methods. Observe that even by doing only a small fraction ($F = 10$ or 10%) of matrix-vector products in reliable mode, the convergence of CG-SS degrades relative to ERROR FREE CG by no more than twice the number of unreliable matrix-vector products at the same accuracy level. For K3D, CG-SS is much better than CG-RES; in the other two cases, it is comparable. FT-GMRES outperforms CG-SS on THERMAL1, where its extra preconditioning confers an advantage, but underperforms on DIAG.

Next consider the right column of Figure 8.1, in which we include exponent bit flips, i.e., *unbounded* errors. (Note that the scale on the x-axes changes from the first row.) As expected, the number of iterations as measured by unreliable matrix-vector products increases, but the qualitative observations of the left column largely hold. As such, we see CG-SS has the potential to withstand unbounded faults.

Amount of reliable computation required

For all the algorithms considered, we need to assume some degree of reliable—and presumably more expensive—computation. This section assesses the number of reliable matrix-vector products required relative to the total number of matrix-vector products. ERROR FREE CG, which computes entirely in reliable mode, provides an upper-bound.

Figure 8.2 shows the fraction of reliable computation needed to achieve the given error tolerance for unbounded errors for K3D problem. (The lower-bound on number of reliable computations is $1/F$ where F is the correction step frequency.) The bit flip errors are the same as in the experiments above. For unbounded errors, CG-SS needs to run just 30% of the upper bound on reliable matvecs that ERROR FREE CG needs; and for bounded errors, just 20% (not shown here). In both cases, the fraction seems to approach some asymptotic limit. Though not shown explicitly here, it can be inferred from Figure 8.1c and Figure 8.1e that CG-SS is only very slightly above very slightly above the lower bound of 10% when $F = 10$. Below, we will see that this bound can be even lower as the fault rate decreases.

8.4 Analysis

We may derive an analytical relationship among correction step frequency, fault rate, and properties of the linear system when errors are bounded (mantissa and sign bit flips). Our analysis is conservative as it invokes coarse bounds; however, it may still be useful in guiding the choice of parameters. The key tool in the analysis is that bounded faults allow the use of results from inexact Krylov subspace methods [75, 78].

Let ΔA_i be the perturbation to A due to bit flips in iteration i of CG-SS. Then,

$$q_i = (A + \Delta A_i)p_i, \quad x_{i+1} = x_i + \alpha_i p_i, \quad \text{and } r_{i+1} = r_i - \alpha_i q_i,$$

where $\alpha_i = \frac{||r_i||^2}{p_i^T q_i}$. The perturbation means $r_{i+1} \neq b - Ax_{i+1}$ as it would without faults. As such, let the true residual be $\hat{r_i} = b - Ax_i$. Then, one can show inductively,

$$w_i = r_i - \hat{r_i} = r_i - (b - Ax_i) = \sum_{j=0}^{i-1} \alpha_j \Delta A_j p_j. \tag{8.9}$$

Suppose we write $r_i = \tilde{b} - Ax_i$, where $\tilde{b} = b - w_i$. Then, CG is effectively solving $Ax = \tilde{b}$ and the residual $\|r_i\|_2$ will continue to decrease, in the absence of faults. (We observed this fact empirically as well.) Hence, $\lim_{i\to\infty} \|r_i\|_2 = 0$. Consequently, from Equation (8.9),

$$\lim_{i\to\infty} \hat{r_i} \approx \sum_{j=0}^{i-1} \alpha_j \Delta A_i p_i = w_i. \tag{8.10}$$

If ΔA_i is bounded, then $||w_i||$ will start from 0 and then quickly increase toward a constant. This limiting value will depend on the bit flip rate.

While $\lim_{i\to\infty} ||r_i|| = 0$, the residual $\hat{r}_i = b - Ax_i$ stagnates. Thus, even if the algorithm converges to the required tolerance, $\hat{r}_i$ might not be close to the solution. The convergence rate resembles that of fault-free execution until $||\hat{r}_i|| \leq ||w_i||$, at which point $||\hat{r}_i||$ stops decreasing.

The analysis has an important implication: the correction step should occur *before* the i at which $||\hat{r}_i|| = ||w_i||$. How can we determine this critical i?

We start by assuming a linear convergence model as $\frac{||r_i||}{||r_0||} = \zeta^i$ for some ζ. We may bound the quantities in Equation (8.10) by

$$\|w_i\|_2 \leq \sum_{j=0}^{i-1} \alpha_j \left\|\Delta A_j p_j\right\|_2.$$

Consider $\alpha_j \left\|\Delta A_j p_j\right\|_2 = \frac{\|r_j\|_2^2}{p_j^T (A+\Delta A_j) p_j} \left\|\Delta A_j p_j\right\|_2$. First,

$$\left\|\Delta A_j p_j\right\|_2 \leq \left\|\Delta A_j\right\|_2 \left\|p_j\right\|_2 \leq \|\Delta A\|_F \left\|p_j\right\|_2,$$

where $\|..\|_F$ denotes the Frobenius norm. Suppose E is the set of non-zero entries in

ΔA_j. If η is number of bit flips in the matrix, then $|E| = \eta$ and

$$\left\|\Delta A_j\right\|_F^2 \leq \sum_{(i,j)\in E} a_{i,\,j}^2 \leq \eta(\max_{(i,\,j)}(a_{i,\,j}))^2$$

Let λ_1 denote the largest eigenvalue of the matrix A. Then, $\max_{i,j}(a_{i,j}) \leq \lambda_1$ and $||\Delta A_j||_F \leq \sqrt{\eta}\lambda_1$. Furthermore, $\left\|r_j\right\|_2 \leq \left\|p_j\right\|_2$ and $p_j^T(A+\Delta A_j)p_j \approx p_j^T A p_j \geq \lambda_n \left\|p_j\right\|^2$, where λ_n is smallest eigenvalue of A. Though dropping ΔA_j may seem arbitrary, we verified empirically that $|\frac{p_j^T \Delta A_j p_j}{p_j^T A p_j}| < 10^{-2}$ on K3D, even at an extremely high fault rate of 400 bit flips per matvec. Combining these expressions, we obtain

$$\alpha_j \left\|\Delta A_j p_j\right\|_2 \leq \sqrt{\eta}\left\|r_j\right\|_2 \frac{\lambda_1}{\lambda_n} = \sqrt{\eta}\left\|r_j\right\|_2 \kappa(A), \tag{8.11}$$

where $\kappa(A)$ denotes the condition number of A. Substituting Equation (8.11) into Equation (8.10) yields

$$\|w_i\|_2 \leq \sum_{j=0}^{i-1} \alpha_j \left\|\Delta A_j p_j\right\|_2 \leq \kappa(A)\sqrt{\eta}\sum_{j=0}^{i}\left\|r_j\right\|_2. \tag{8.12}$$

If r_i converges linearly, then there is a ζ_1 such that $\frac{||r_i||}{||r_0||} \leq \zeta_1^i$. Then, Equation (8.12) becomes

$$\begin{aligned} \|w_i\|_2 &\leq \kappa(A)\sqrt{\eta}(\sum_{j=0}^{i-1}\left\|r_j\right\|_2) \\ &\leq \kappa(A)\sqrt{\eta}\,\|r_0\|_2/(1-\zeta_1) = C\cdot\kappa(A)\sqrt{\eta}\,\|r_0\|_2. \end{aligned} \tag{8.13}$$

for some constant C. Section 8.4 bounds $\|w_i\|$ from above. The critical iteration i occurs when $||\hat{r}_i|| = \mathcal{O}(||w_i||)$; since $||\hat{r}_i|| = \|r_0\|\zeta^i$, we conclude from Section 8.4 that

$$i = \mathcal{O}\left(\log_\zeta\left(\kappa(A)\sqrt{\eta}\right)\right). \tag{8.14}$$

We can see the key scaling relations among i, $\kappa(A)$, and η: i depends on the number of bit flips, η, logarithmically. Thus, doubling the bit flip rate reduces i by just $-\frac{1}{2}\log_\zeta 2$.

Figure 8.3 verifies this analysis on K3D. Specifically, recall that Equation (8.14) predicts the minimum correction frequency needed to attain a desired *convergence profile*, i.e., decrease in the residual as iterations increase. Thus, we use Equation (8.14) to compute i and set the correction frequency F in CG-SS to it.[3] We then observe the actual convergence profiles as the fault rate increases. Each line in Figure 8.3 corresponds to a different value of η and the corresponding F predicted by Equation (8.14).

These results reveal two attractive aspects of CG-SS. First, Figure 8.3a suggests that Equation (8.14) is conservative: the convergence actually *improves* with a decrease in fault rate, rather than staying the same. This aspect of CG-SS is attractive because it implies CG-SS can in principle *gracefully adapt* to the fault rate. Secondly, the overhead also gracefully decreases as the fault rate decreases. Figure 8.3b shows that as the fault rate decreases, if we adapt the correction frequency we can reduce the fraction of reliable matrix-vector products required. Taken together, these observations point to an important direction for future work, namely, how to exploit adaptivity automatically.

Lastly, observe that as the error tolerance limit decreases, CG-SS stagnates, e.g., green squares in Figure 8.1. This behavior depends on the problem and may or may not be acceptable. It suggests we need deeper analysis of the relationship between numerical properties and stability of CG-SS and its ilk. We leave this possibility as future work.

[3] We analyzed the fault-free case to chose constants, e.g., $\zeta = 0.74$.

8.5 Related Work

Our work is most directly inspired by the FT-GMRES method of Hoemmen and Heroux [66], which we included in our comparisons of Section 8.3. Other than our focus on the SPD case in this paper, we view both CG-SS and FT-GMRES as instances of self-stabilization. We believe the more direct connection to the lens of self-stabilization adds a new dimension to this class of methods.

Complementary to our approach, there is a large and growing literature on *algorithm-based fault-tolerance* (ABFT) techniques, primarily by testing checksum invariants [79, 80, 81, 82, 83], including for preconditioned CG [84]. These work well when faults are very small. Zizhong et al [85] use orthogonality relations instead of checksums. However, the cost of this method grows quickly with increasing fault rates. The theory of inexact Krylov subspace methods suggests that many of these methods could further exploit other intrinsic solver properties, to reduce the amount of recomputation involved after detecting an error [78, 75, 76]. Since self-stabilizing does not use fault detection, future work could study hybrid ABFT and self-stabilizing methods.

Oboril et al. suggest doing CG iterations on the residual if convergence slows or stops due to faults [86]. Their method is essentially CG-RES, against which we compared.

Lastly, there is a long history of asynchronous iteration techniques [87, 88]. However, these are primarily based on stationary iteration, whereas Krylov methods tend to have more favorable convergence rates and (in exact arithmetic) finite termination.

8.6 Conclusion and Future work

Self-stabilization expands our collective repertoire of resilience techniques. We believe the abstraction of state transitions and correction steps is a useful lens for viewing the problem of how to design resilient solvers. Furthermore, by formally

connecting the problem to the "orthogonal" literature of self-stabilizing algorithms may permit many new and creative approaches to the design of resilient solvers broadly [69]. We sketch some directions below.

A tighter and broader model of convergence. The analysis of Section 8.4 is useful but coarse, being limited to easily bounded errors. Exploring this problem and others, in a more deeply theoretically way, is an important direction for future work. For instance, the theory of inexact Krylov methods includes results that permit errors in each matrix-vector product to gradually *increase* [78, 75], in part because errors in later iterations have a reduced impact. This theory might permit adaptive correction step frequencies, for instance. In addition, the numerical stability and convergence rate properties of our CG-SS method is only partly understood.

The balance between reliable and unreliable computation. One major finding of our study is the tradeoff between reliable and unreliable computation. This raises numerous questions, such as whether there is a fundamental lower bound on amount of reliable computation. Additionally, we do not yet have a good model of the difference in the time, energy, and power *costs* between reliable and unreliable mode, as has been suggested in numerous hardware and circuit-level analyses [77, 86].

Self-stabilizing other algorithms. Another direction for future work is to design self-stabilized solvers under preconditioning or for other solvers altogether. We expect the results of this paper extend to other CG-like methods, such as biconjugate gradients (BiCG), BiCGStab, CG-squared, and Chebyshev iterations, among others [89].

Hybridized fault-tolerance techniques. As noted in Section 8.3.4, self-stabilization may fruitfully *complement* ABFT and other techniques. Such an approach may be useful for unbounded faults, as occur with exponent bit flips.

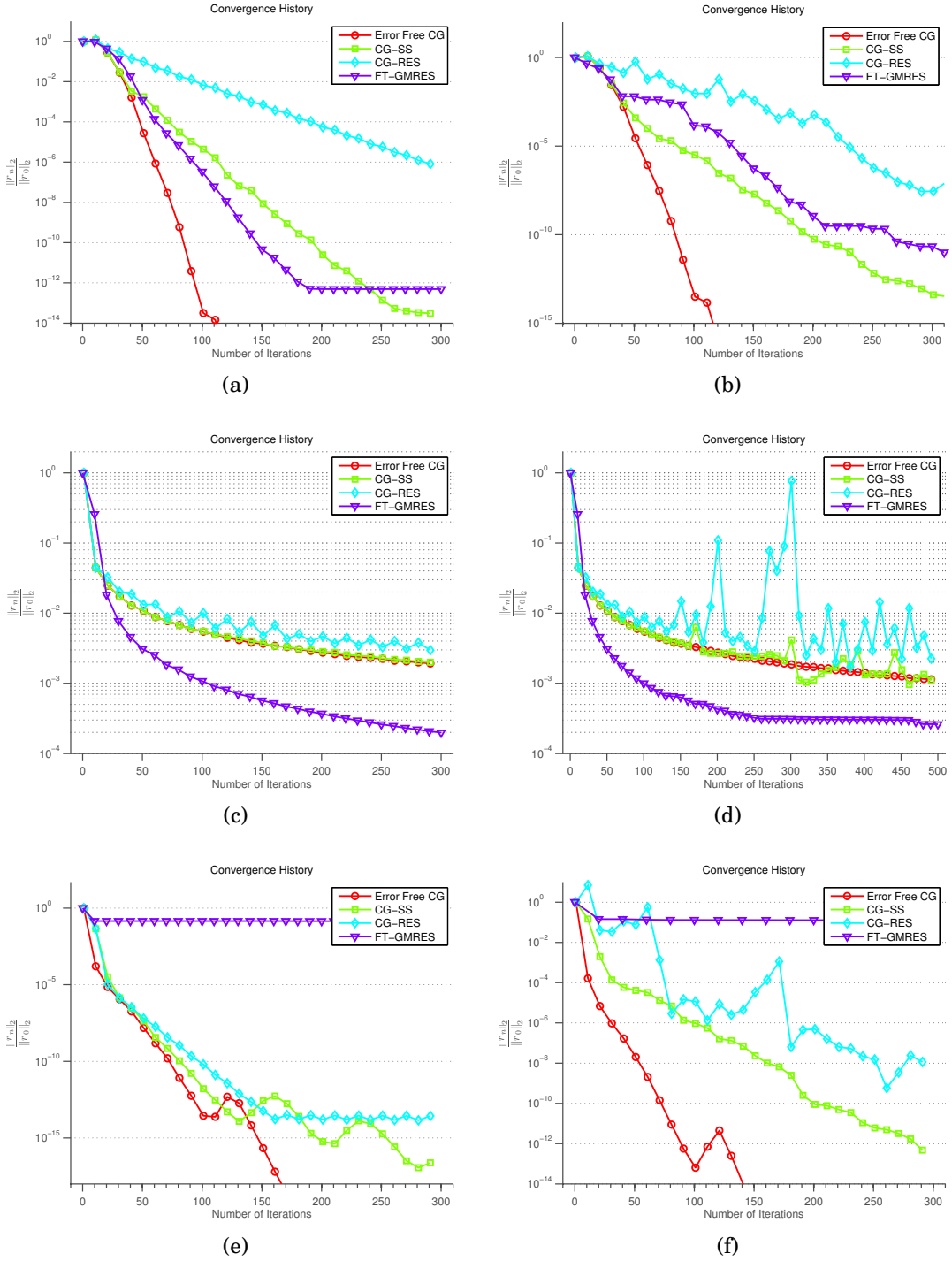
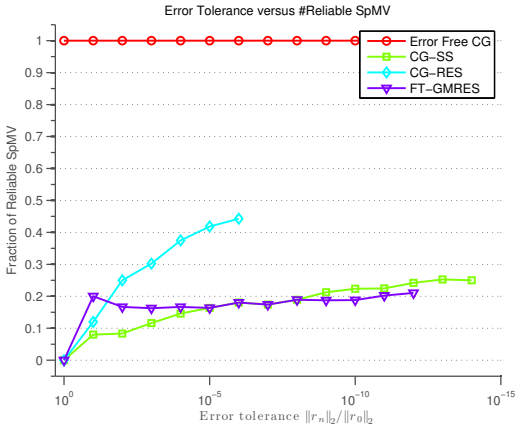
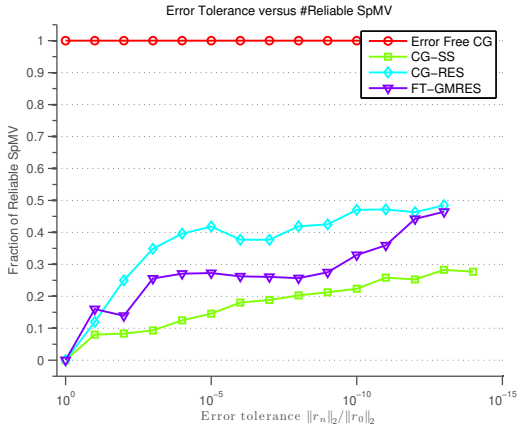


Figure 8.1: This figure shows the convergence history for different problems: K3D (first row), THERMAL1 (second row), and DIAG (last row). The left plots include only sign and mantissa bit flips at a rate of 40 bit flips in every unreliable matrix-vector product, while the right plots include exponent bit flips at a rate of 4 bit flips per unreliable matvec.

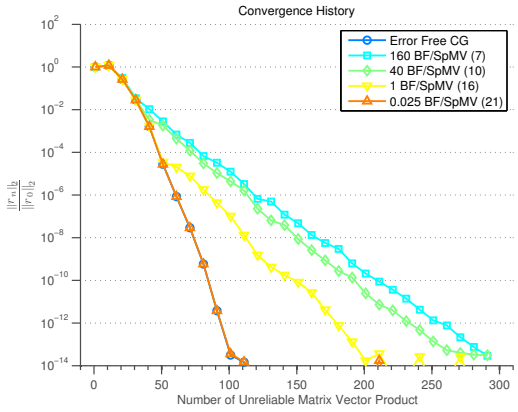


(a) Bounded errors (mantissa and sign bit flips only)

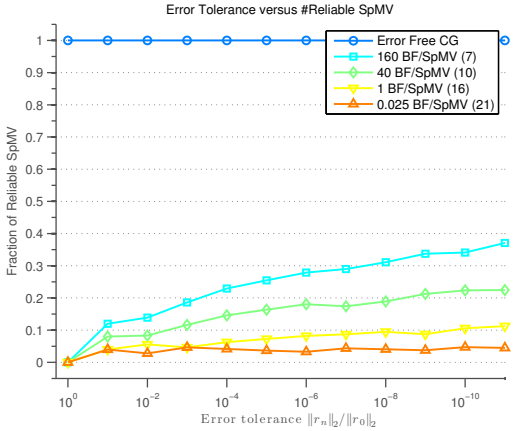


(b) Unbounded errors (including exponent bit flips)

Figure 8.2: The fraction of matrix-vector products (with respect to matrix-vector products required by ERROR FREE CG) must run in reliable mode (y-axis) to achieve some accuracy (x-axis).



(a) Convergence history for different fault rates



(b) Number of reliable (sparse) matrix-vector products ("SpMV") for different fault rates

Figure 8.3: Comparison of the predicted correction step frequency to the observed behavior of CG-SS at varying fault rates (different lines). "BF" = bit flips; "SpMV" = (sparse) matrix-vector product. For each line, the value in parentheses is the estimated critical i from Equation (8.14) at the corresponding bit flip rate.

CHAPTER 9

CONCLUSION AND FUTURE DIRECTIONS

> Prediction is very difficult, especially about the future.
>
> Niels Bohr

9.1 Summary

We summarize the main contribution and results of this thesis below.

9.1.1 GPU and Xeon-Phi accelerated Distributed Memory Sparse Direct Solver

The first problem that we considered was how to explore co-processors such as GPU and Xeon-Phi effectively for the distributed memory sparse direct solver. As a first step towards realizing this goal, we used GPUs as the dense BLAS-accelerator. This approach while effective on older architecture didn't achieve what was ultimately possible. We extended to the of co- processors to perform most of the Schur-complement update computations. We presented a new offload algorithm HALO to do so. The HALO algorithm was better than our previous approach in two ways. First, it offloaded SCATTER computations in addition to GEMM computations. And second, it reduced the PCIe communication between the host multicore and the co-processor significantly. We presented an efficient implementation of the HALO algorithm, initially for MIC accelerated systems, and later for GPU accelerated clusters. We addressed the issue of load balancing between the co- processors and host; and how to effectively use HALO when the co-processors have much smaller memory than the host. Overall, with GPU and MIC acceleration SUPERLU_DIST can achieve up to 3×

speed-up relative to a highly optimized multicore implementation.

9.1.2 A 3D LU factorization algorithm for sparse matrices

We presented a new algorithm that significantly improved the strong scalability of the state-of-the-art distributed memory sparse direct solver. In contrast to the previous approaches, which focused on hiding the communication costs, our algorithm reduces the total communication volume by a factor of $\mathcal{O}\left(\sqrt{\log n}\right)$ for a problem of the dimension of n, in the numerical factorization step of the sparse direct solver. The new 3D algorithm is up to 27× faster for sparse matrix when the associated graph is planar, and up to 3× faster for sparse matrices with 3D geometry, while scalably using up to 24,000 cores of Cray XK7 machine.

9.1.3 Self-stabilizing iterative solvers

We applied the principle of self-stabilization to construct sparse iterative solvers that can tolerate high injected fault rates. Specifically, we presented the analysis of valid and invalid states of conjugated gradient algorithm. We designed a novel correction step that can bring the conjugated gradient algorithm from an invalid state to a guaranteed state. We presented the convergence analysis of the conjugate-gradient algorithm in presence of the bit-flips type of fault and presented bounds on the number of computations that should be done reliably for the conjugate-gradient algorithm to converge in presence of faults.

9.2 Future Directions

Energy Efficient Algorithms The amount of energy a solver algorithm takes is the *running* cost of solving a sparse system. Thus, future algorithm designers must consider the energy costs while designing a solver algorithm. We indirectly address this issue by exploiting energy efficient co-processors and reducing inter and

intra-node communication and reducing computation replication by using the fault-tolerant algorithms. However, in the near future, we need to address the issue of directly. To make such designing possible, researchers have proposed accessible cost models for power and energy of the algorithm execution [90]. Developing performance models for sparse solvers that also includes the power and energy costs, is the next step to the eventual goal of designing power and energy efficient sparse linear solvers.

Near Memory Computing A recurring problem in sparse linear algebra is irregular and indirect memory access pattern which are not efficient on traditional hierarchical memory systems. Off-chip communication can be significantly more expansive than on-chip communication.

From a hardware perspective, this challenge is addressed in following two ways. First is stacking DRAM cells on the same die as the processor core by using 3D stacked fabrication technology, known as High Bandwidth Memory (HBM).

The second approach is to add a logic layer on the Dynamic Random-access Memory (DRAM) wafer itself, so as to perform the operations with lower arithmetic intensity on the DRAM chip itself.

These hardware innovations can improve the performance of SCATTER and gather computations relative to traditional memory systems.

Thus probing such near memory accelerators for improving the performance of sparse linear solvers is worth exploring.

Approximate Computing Approximate computing aims to trade-off accuracy for reduced resources, such as time or energy. Our work on self-stabilizing iterative solvers suggests that by doing as little as 10% without errors, we can still obtain the desired level of precision for numerical solvers. Thus, we have initial point results suggesting the usefulness of such a paradigm for a more balanced resource utiliza-

tion.

Sparse Solvers with Lower Latency The latency costs of an algorithm present the fundamental limits on how fast the execution can be. In case of sparse solvers, the latency cost can be due to programmability, or the algorithm or by the underlying hardware. The unavailability of programming models that can easily express the parallelism in the solver algorithm and translate to such execution, can add to avoidable latencies. We expect by adapting to newer programming models such as MPI's one-sided communication, inter GPU communications, we can further push the latency limits. There are some latencies which are inherent to the algorithm. One such example is the depth of dense LU factorization, which is $\mathcal{O}(n)$ as oppose to $\mathcal{O}(\log n)$ for matrix multiplication. The variant of LU factorization that can break such barriers will provide a big jump on the scalability of direct methods. Yet, if such a technique is possible or not is an open question.

REFERENCES

[1] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration", in *Proceedings of the 2nd usenix conference on hot topics in parallelism*, USENIX Association, 2010, pp. 13–13.

[2] O. Schenk, A. Wächter, and M. Weiser, "Inertia-revealing preconditioning for large-scale nonconvex constrained optimization", *Siam journal on scientific computing*, vol. 31, no. 2, pp. 939–960, 2008.

[3] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems", *Acm transactions on mathematical software (toms)*, vol. 37, no. 3, p. 36, 2010.

[4] T. A. Davis, *Direct methods for sparse linear systems*. Siam, 2006, vol. 2.

[5] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Clarendon press Oxford, 1986.

[6] G. Karypis and V. Kumar, *Family of graph and hypergraph partitioning software*, `http://glaros.dtc.umn.edu/gkhome/views/metis`, Accessed: 2014-01-26.

[7] E. Rothberg, "Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization", STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1992.

[8] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel algorithms for sparse linear systems", *Siam review*, vol. 33, no. 3, pp. 420–460, 1991.

[9] J. H. Wilkinson, *Rounding errors in algebraic processes*. Courier Corporation, 1994.

[10] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface", *Acm transactions on mathematical software (toms)*, vol. 31, no. 3, pp. 302–325, 2005.

[11] I. Yamazaki and X. S. Li, "New scheduling strategies and hybrid programming for a parallel right-looking sparse lu factorization algorithm on multicore cluster systems", in *Parallel & distributed processing symposium (ipdps), 2012 ieee 26th international*, IEEE, 2012, pp. 619–630.

[12] P. Sao, R. Vuduc, and X. S. Li, "A distributed cpu-gpu sparse direct solver", in *European conference on parallel processing*, Springer, 2014, pp. 487–498.

[13] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems", *Acm trans. mathematical software*, vol. 29, no. 2, pp. 110–140, 2003.

[14] *MAGMA – Matrix Algebra on GPU and Multicore Architectures*, `http://icl.utk.edu/magma/`.

[15] G. Krawezik and G. Poole, "Accelerating the ANSYS direct sparse solver with GPUs", in *Proc. symposium on application accelerators in high performance computing (saahpc)*, `http://saahpc.ncsa.illinois.edu/09`, Urbana-Champaign, IL, NCSA, 2009.

[16] C. Yu, W. Wang, and D. Pierce, "A CPU-GPU hybrid approach for the unsymmetric multifrontal method", *Parallel computing*, vol. 37, pp. 759–770, 2011.

[17] O. Schenk, M. Christen, and H. Burkhart, "Algorithmic performance studies on graphics processing units", *J. parallel and distributed computing*, vol. 68, no. 10, pp. 1360–1369, 2008.

[18] T. George, V. Saxena, A. Gupta, A. Singh, and A. Choudjury, "Multifrontal factorization of sparse spd matrices on GPUs", in *Proc. of ieee international parallel and distributed processing symposium (ipdps 2011)*, Anchorage, Alaska, 2011.

[19] R. Lucas, G. Wagenbreth, D. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts", in *Vecpar'10: Proc. 9th intl. meeting for high performance computing for computational science*, `http://vecpar.fe.up.pt/2010/papers/5.php`, Berkeley, CA, 2010.

[20] S. Yeralan, T. Davis, and S. Ranka, "Sparse QR factorization on GPU architectures", University of Florida, Tech. Rep., 2013.

[21] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix ollection", *Acm transactions on mathematical software (toms)*, vol. 38, no. 1, p. 1, 2011.

[22] *Ipm : Integrated performance monitoring*, `http://ipm-hpc.sourceforge.net/`, Accessed: 2014-01-26.

[23] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems", *Acm trans. mathematical software*, vol. 29, no. 2, pp. 110–140, 2003.

[24] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration", in *Proc. of the 2nd usenix conference on hot topics in parallelism, hotpar'10*, Berkeley, CA, 2010.

[25] G. Krawezik and G. Poole, "Accelerating the ANSYS direct sparse solver with GPUs", in *Proc. symposium on application accelerators in high performance computing (saahpc)*, Urbana-Champaign, IL, NCSA, 2009.

[26] T. George, V. Saxena, A. Gupta, A. Singh, and A. Choudjury, "Multifrontal factorization of sparse SPD matrices on GPUs", in *Proc. of ieee international parallel and distributed processing symposium (ipdps 2011)*, Anchorage, Alaska, 2011.

[27] R. Lucas, G. Wagenbreth, D. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts", in *Vecpar'10: Proc. 9th intl. meeting for high performance computing for computational science*, Berkeley, CA, 2010.

[28] C. Yu, W. Wang, and D. Pierce, "A CPU-GPU hybrid approach for the unsymmetric multifrontal method", *Parallel computing*, vol. 37, pp. 759–770, 2011.

[29] P. Sao, X. Liu, R. Vuduc, and X. Li, "A sparse direct solver for distributed memory xeon phi-accelerated systems", in *Parallel and distributed processing symposium (ipdps), 2015 ieee international*, IEEE, 2015, pp. 71–81.

[30] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms", in *Euro-par 2011 parallel processing*, Springer, 2011, pp. 90–109.

[31] P. Sao, R. Vuduc, and X. S. Li, "A distributed CPU-GPU sparse direct solver", in *Euro-par 2014 parallel processing*, Springer International Publishing, 2014, pp. 487–498.

[32] M. Hoemmen, *Communication-avoiding krylov subspace methods*. University of California, Berkeley, 2010.

[33] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang, "Communication avoiding rank revealing qr factorization with column pivoting", *Siam journal on matrix analysis and applications*, vol. 36, no. 1, pp. 55–89, 2015.

[34] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations", in *Parallel and distributed processing, 2008. ipdps 2008. ieee international symposium on*, IEEE, 2008, pp. 1–12.

[35] D. Irony and S. Toledo, "Trading replication for communication in parallel distributed-memory dense solvers", *Parallel processing letters*, vol. 12, no. 01, pp. 79–94, 2002.

[36] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs", *Siam journal on applied mathematics*, vol. 36, no. 2, pp. 177–189, 1979.

[37] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan, "A separator theorem for graphs of bounded genus", *Journal of algorithms*, vol. 5, no. 3, pp. 391–407, 1984.

[38] N. Alon, P. Seymour, and R. Thomas, "A separator theorem for graphs with an excluded minor and its applications", in *Proceedings of the twenty-second annual acm symposium on theory of computing*, ACM, 1990, pp. 293–299.

[39] C. Ashcraft, "A taxonamy of distributed dense LU factorization methods", *Engineering computing and analysis technical report eca-tr-161, boeing computer services*, 1991.

[40] C. Ashcraft, "The fan-both family of column-based distributed Cholesky factorization algorithms", in *Graph theory and sparse matrix computation*, Springer, 1993, pp. 159–190.

[41] L. Hulbert and E. Zmijewski, "Limiting communication in parallel sparse Cholesky factorization", *Siam journal on scientific and statistical computing*, vol. 12, no. 5, pp. 1184–1197, 1991.

[42] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization", *Ieee transactions on parallel and distributed systems*, vol. 8, no. 5, pp. 502–520, 1997.

[43] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication", *Journal of parallel and distributed computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[44] G. M. Ballard, "Avoiding communication in dense linear algebra", PhD thesis, 2013.

[45] H. Jia-Wei and H.-T. Kung, "I/o complexity: The red-blue pebble game", in *Proceedings of the thirteenth annual acm symposium on theory of computing*, ACM, 1981, pp. 326–333.

[46] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet, "Robust memory-aware mappings for parallel multifrontal factor-

izations", *Siam journal on scientific computing*, vol. 38, no. 3, pp. C256–C279, 2016.

[47] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel scheduling of task trees with limited memory", *Acm transactions on parallel computing*, vol. 2, no. 2, p. 13, 2015.

[48] A. Khabou, J. Demmel, L Grigori, and M Gu, "Communication avoiding lu factorization with panel rank revealing pivoting", *Siam journal on matrix analysis and applications*, vol. 34, no. 3, pp. 1401–1429, 2013.

[49] L. Grigori, J. W. Demmel, and H. Xiang, "Calu: A communication optimal lu factorization algorithm", *Siam journal on matrix analysis and applications*, vol. 32, no. 4, pp. 1317–1350, 2011.

[50] M. Baboulin, X. S. Li, and F.-H. Rouet, "Using random butterfly transformations to avoid pivoting in sparse direct methods", in *International conference on high performance computing for computational science*, Springer, 2014, pp. 135–144.

[51] K. Kim and V. Eijkhout, "A parallel sparse direct solver via hierarchical dag scheduling", *Acm transactions on mathematical software (toms)*, vol. 41, no. 1, p. 3, 2014.

[52] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes", in *Parallel & distributed processing symposium workshops (ipdpsw), 2014 ieee international*, IEEE, 2014, pp. 29–38.

[53] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems", *Acm transactions on mathematical software (toms)*, vol. 43, no. 2, p. 13, 2016.

[54] E. Agullo, L. Giraud, and S. Nakov, "Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures", in *European conference on parallel processing*, Springer, 2016, pp. 83–95.

[55] M. Jacquelin, Y. Zheng, E. Ng, and K. Yelick, "An asynchronous task-based fan-both sparse Cholesky solver", *Arxiv preprint arxiv:1608.00044*, 2016.

[56] D. Goik, K. Jopek, M. Paszyński, A. Lenharth, D. Nguyen, and K. Pingali, "Graph grammar based multi-thread multi-frontal direct solver with galois scheduler", *Procedia computer science*, vol. 29, pp. 960–969, 2014.

[57] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, “Optimized hpl for amd gpu and multi-core cpu usage.”, *Computer science-r&d*, vol. 26, no. 3-4, pp. 153–164, 2011.

[58] P. Strazdins *et al.*, “A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization”, 1998.

[59] X. Lacoste, “Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems”, PhD thesis, Bordeaux, 2015.

[60] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, “A runtime approach to dynamic resource allocation for sparse direct solvers”, in *Parallel processing (icpp), 2014 43rd international conference on*, IEEE, 2014, pp. 481–490.

[61] *Top500 list*, http://www.top500.org/, 2013.

[62] G. Bronevetsky and B. de Supinski, “Soft error vulnerability of iterative linear algebra methods”, in *Proceedings of the 22nd annual international conference on supercomputing*, ACM, 2008, pp. 155–164.

[63] B. Schroeder, E. Pinheiro, and W.-D. Weber, “Dram errors in the wild: A large-scale field study”, in *Acm sigmetrics performance evaluation review*, ACM, vol. 37, 2009, pp. 193–204.

[64] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, “On the optimal design of triple modular redundancy logic for SRAM-based FPGAs”, in *Proceedings of the conference on design, automation and test in europe-volume 2*, IEEE Computer Society, 2005, pp. 1290–1295.

[65] ——, “On the optimal design of triple modular redundancy logic for SRAM-based FPGAs”, in *Proceedings of the conference on design, automation and test in europe-volume 2*, IEEE Computer Society, 2005, pp. 1290–1295.

[66] M. Hoemmen and M. A. Heroux, “Fault-tolerant iterative methods via selective reliability”, in *Proceedings of the 2011 international conference for high performance computing, networking, storage and analysis (sc). ieee computer society*, vol. 3, 2011, p. 9.

[67] R. Guerraoui and A. Schiper, “Software-based replication for fault tolerance”, *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[68] E. W. Dijkstra, “Self-stabilization in spite of distributed control”, in *Selected writings on computing: A personal perspective*, Springer, 1982, pp. 41–46.

[69] S. Dolev, *Self-stabilization*. MIT press, 2000.

[70] C. Kelley, "Iterative methods for linear and nonlinear equations", *Mr 96d*, vol. 65002, 1995.

[71] J. R. Shewchuk *et al.*, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[72] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems", *Siam journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[73] G. Zoutendijk, *Methods of feasible directions: A study in linear and non-linear programming*. Elsevier, 1960.

[74] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, "Improving the performance of dynamical simulations via multiple right-hand sides", in *Parallel & distributed processing symposium (ipdps), 2012 ieee 26th international*, IEEE, 2012, pp. 36–47.

[75] J. Van Den Eshof and G. L. Sleijpen, "Inexact Krylov subspace methods for linear systems", *Siam journal on matrix analysis and applications*, vol. 26, no. 1, pp. 125–153, 2004.

[76] G. H. Golub and Q. Ye, "Inexact preconditioned conjugate gradient method with inner-outer iteration", *Siam journal on scientific computing*, vol. 21, no. 4, pp. 1305–1320, 1999.

[77] A. Gupta, S. Mandavalli, V. J. Mooney, K.-V. Ling, A. Basu, H. Johan, and B. Tandianus, "Low power probabilistic floating point multiplier design", in *Vlsi (isvlsi), 2011 ieee computer society annual symposium on*, IEEE, 2011, pp. 182–187.

[78] V. Simoncini and D. B. Szyld, "Theory of inexact Krylov subspace methods and applications to scientific computing", *Siam journal on scientific computing*, vol. 25, no. 2, pp. 454–477, 2003.

[79] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations", *Ieee transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[80] C. J. Anfinson and F. T. Luk, "A linear algebraic model of algorithm-based fault tolerance", *Ieee transactions on computers*, vol. 37, no. 12, pp. 1599–1604, 1988.

[81] Y.-M. Yeh and T.-y. Feng, “Algorithm-based fault tolerance for matrix inversion with maximum pivoting”, *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 373–389, 1992.

[82] F. T. Luk and H. Park, “An analysis of algorithm-based fault tolerance techniques”, *Journal of parallel and distributed computing*, vol. 5, no. 2, pp. 172–184, 1988.

[83] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, “Algorithm-based fault tolerance for dense matrix factorizations”, *Acm sigplan notices*, vol. 47, no. 8, pp. 225–234, 2012.

[84] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, “Fault tolerant preconditioned conjugate gradient for sparse linear system solution”, in *Proceedings of the 26th acm international conference on supercomputing*, ACM, 2012, pp. 69–78.

[85] Z. Chen, “Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods”, in *Acm sigplan notices*, ACM, vol. 48, 2013, pp. 167–176.

[86] F. Oboril, M. B. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss, “Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers”, in *Dependable computing (prdc), 2011 ieee 17th pacific rim international symposium on*, IEEE, 2011, pp. 144–153.

[87] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel iterative algorithms: From sequential to grid computing*. CRC Press, 2007.

[88] H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline, “A block-asynchronous relaxation method for graphics processing units”, *Journal of parallel and distributed computing*, vol. 73, no. 12, pp. 1613–1626, 2013.

[89] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, 1994.

[90] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, “A roofline model of energy”, in *Parallel & distributed processing (ipdps), 2013 ieee 27th international symposium on*, IEEE, 2013, pp. 661–672.